

ASP.NET CORE

Essentials for Beginners 100



2024 Latest Edition

Index

[ASP.NET Core is Cross-Platform](#)
[ASP.NET Core Supports Dependency Injection](#)
[Understanding Middleware in ASP.NET Core](#)
[Kestrel: The Default Web Server for ASP.NET Core](#)
[Razor Pages: A Simpler Alternative to MVC in ASP.NET Core](#)
[Unified Web API and MVC Framework in ASP.NET Core](#)
[Understanding Configuration Hierarchy in ASP.NET Core](#)
[Caching Strategies in ASP.NET Core](#)
[Entity Framework Core for Database Interactions](#)
[ASP.NET Core Identity for User Authentication](#)
[Understanding Tag Helpers in ASP.NET Core](#)
[Logging in ASP.NET Core with Built-in Support](#)
[Health Checks in ASP.NET Core for Application Monitoring](#)
[Environment-Specific Behavior in ASP.NET Core Hosting Environment](#)
[Hosting Options for ASP.NET Core Applications](#)
[Understanding Minimal APIs in ASP.NET Core](#)
[Global Error Handling in ASP.NET Core with UseExceptionHandler](#)
[Middleware](#)
[High-Performance Remote Procedure Calls with gRPC in ASP.NET Core](#)
[Understanding appsettings.json in ASP.NET Core](#)
[Introduction to Action Filters in ASP.NET Core](#)
[Data Protection in ASP.NET Core: Encrypting and Decrypting Data](#)
[SignalR in ASP.NET Core: Real-Time Web Functionality](#)
[Understanding Model Binding in ASP.NET Core](#)
[Creating Custom Middleware in ASP.NET Core](#)
[Customizable Routing in ASP.NET Core](#)
[Building Web UIs with Blazor and C#](#)
[Configuring the Request Pipeline with IApplicationBuilder](#)
[Understanding the IHostingEnvironment Interface](#)
[Default HTTPS Support in ASP.NET Core](#)
[Deployment Options in ASP.NET Core](#)

[Built-in Support for Data Formats in ASP.NET Core APIs](#)
[Serving Static Files with Middleware](#)
[Customizing the Dependency Injection \(DI\) Container in ASP.NET Core](#)
[Handling Multiple Cultures and Languages with Request Localization in ASP.NET Core](#)
[Understanding Razor Syntax in ASP.NET Core](#)
[Preventing CSRF Attacks with Anti-Forgery Tokens in ASP.NET Core](#)
[Storing Sessions in ASP.NET Core](#)
[Understanding ControllerBase in ASP.NET Core](#)
[Middleware Execution Order in ASP.NET Core](#)
[Understanding IActionResult in ASP.NET Core](#)
[API Versioning in ASP.NET Core](#)
[Logging with ILogger in ASP.NET Core](#)
[Running ASP.NET Core Applications in Docker Containers](#)
[Understanding ModelState for Input Validation in ASP.NET Core](#)
[Handling File Uploads with IFormFile in ASP.NET Core](#)
[Using UserManager for User Operations in ASP.NET Core Identity](#)
[Introduction to Razor View Engine in ASP.NET Core](#)
[ASP.NET Core Authentication Middleware Overview](#)
[Authorization Policies in ASP.NET Core](#)
[Configuring and Starting a Web Application with WebHostBuilder](#)
[Custom Route Constraints in ASP.NET Core](#)
[Using HttpClientFactory in ASP.NET Core](#)
[Using GraphQL with ASP.NET Core for Flexible APIs](#)
[Building Configuration Sources with ConfigurationBuilder in ASP.NET Core](#)
[Using Environment Variables in ASP.NET Core](#)
[JWT Bearer Tokens for API Authentication in ASP.NET Core](#)
[Understanding Dependency Injection Scopes in ASP.NET Core](#)
[Mastering Model Validation in ASP.NET Core](#)
[Accessing Application Settings with IConfiguration in ASP.NET Core](#)
[Handling Exceptions with Custom Exception Filters in ASP.NET Core](#)
[Understanding View Components in ASP.NET Core](#)

[Using ObjectResult in ASP.NET Core](#)
[Understanding the IFormCollection Interface in ASP.NET Core](#)
[CORS Support in ASP.NET Core](#)
[Using TempData in ASP.NET Core](#)
[Generating URLs with IUrlHelper in ASP.NET Core](#)
[Configuring Session State in ASP.NET Core](#)
[Adding Real-Time Functionality with ASP.NET SignalR](#)
[Response Compression in ASP.NET Core](#)
[Database Configuration by Environment in ASP.NET Core](#)
[Understanding Endpoint Routing in ASP.NET Core](#)
[Managing Forms with ASP.NET Core's FormTagHelper](#)
[File Streaming in ASP.NET Core Using FileStreamResult](#)
[Configuring Cookies with ASP.NET Core's CookieOptions](#)
[Configuring Custom Razor View Locations in ASP.NET Core](#)
[Understanding the ActionResult Class in ASP.NET Core](#)
[Dynamic Compilation of Razor Views in ASP.NET Core](#)
[Implementing Custom Logic with IActionFilter in ASP.NET Core](#)
[Understanding IFormFileCollection in ASP.NET Core](#)
[Standardizing API Responses with API Conventions in ASP.NET Core](#)
[Custom Authorization with AuthorizationHandler in ASP.NET Core](#)
[Global Exception Handling with Middleware in ASP.NET Core](#)
[Razor Pages Simplify Page-Focused Development in ASP.NET Core](#)
[Using StatusCodes Class for HTTP Status Codes in ASP.NET Core](#)
[Using IApplicationLifetime for Application Startup and Shutdown in ASP.NET Core](#)
[Enabling Request Body Buffering in ASP.NET Core](#)
[Understanding ActionResult<T> in ASP.NET Core](#)
[Using HttpContext in ASP.NET Core](#)
[Configuring Form Data Processing with ASP.NET Core's FormOptions Class](#)
[Passing Data to Views Using ViewData in ASP.NET Core](#)
[Graceful Shutdown in ASP.NET Core](#)
[Generating URLs with Url.Action in ASP.NET Core](#)

[Limiting Request Size in ASP.NET Core](#)

[Using ApiController Attribute in ASP.NET Core](#)

[Understanding Cookie Authentication in ASP.NET Core](#)

[Configuring CORS Policies with CorsPolicyBuilder in ASP.NET Core](#)

[Understanding PartialView Method in ASP.NET Core](#)

[Forward Compatibility in ASP.NET Core with Feature Flags](#)

[Using JsonResult to Return JSON Data in ASP.NET Core](#)

[Understanding AssemblyPart in ASP.NET Core MVC](#)

[Understanding the IWebHostEnvironment Interface in ASP.NET Core](#)

[Exploring the DefaultHttpContext Class in ASP.NET Core](#)

Introduction



Welcome to this essential guide designed for those who have a foundational understanding of programming and are ready to delve into the world of ASP.NET Core.

This eBook focuses exclusively on the critical knowledge required for beginners in ASP.NET Core, ensuring that you can acquire only the most relevant information without unnecessary distractions.

Whether you are just starting your journey with ASP.NET Core or looking to refresh your skills, this resource is tailored to meet your needs.

It provides a streamlined approach to learning, allowing you to grasp the essential concepts and practices that will empower you in your development projects.

We also encourage seasoned developers to utilize this book as a quick reference to the latest must-know aspects of ASP.NET Core, making it a versatile tool for all levels of expertise.

Your feedback is invaluable to us! We would greatly appreciate it if you could take a moment to leave a review or comment after reading.

Your insights help us improve and provide even better resources in the future.

Thank you for choosing this guide, and happy coding!

1

ASP.NET Core is Cross-Platform

ASP.NET Core can run on Windows, macOS, and Linux, making it versatile for different development environments.

To demonstrate ASP.NET Core's cross-platform capability, you can create a simple web application and run it on different operating systems.

[Code]

```
# On Windows
dotnet new webapp -o MyWebApp
cd MyWebApp
dotnet run
# On macOS or Linux
dotnet new webapp -o MyWebApp
cd MyWebApp
dotnet run
```

[Result]

When you run `dotnet run`, the application will start, and you can access it through a web browser at `http://localhost:5000` regardless of the operating system you're using.

ASP.NET Core was designed to be cross-platform from the ground up, allowing developers to build and deploy applications on the operating system of their choice. This is particularly useful in environments where developers use different systems or when deploying applications to cloud platforms that may not run Windows. The ability to work seamlessly across Windows, macOS, and Linux ensures that ASP.NET Core fits into diverse development pipelines. .NET Core, the runtime that powers ASP.NET Core, uses platform-agnostic libraries and a modular design. This means the same codebase can be compiled and run on different platforms without modification. The `dotnet CLI`, which is the command-

line interface for .NET, provides a consistent way to build, run, and manage projects across these operating systems, ensuring a uniform development experience.

[Trivia]

Cross-platform capability is achieved using the .NET runtime, which abstracts away the underlying operating system. This allows developers to write code that can run on different platforms without needing to change the code. It is especially beneficial for teams that work in diverse environments or for applications that need to be deployed across multiple platforms.

2

ASP.NET Core Supports Dependency Injection

ASP.NET Core natively supports dependency injection (DI), which is a design pattern that allows for the injection of dependencies into a class.

Let's look at a simple example to understand how dependency injection works in ASP.NET Core by injecting a service into a controller.

[Code]

```
// Define a service interface
public interface IMyService
{
    string GetMessage();
}
// Implement the service
public class MyService : IMyService
{
    public string GetMessage()
    {
        return "Hello, Dependency Injection!";
    }
}
// Register the service in the Startup.cs file
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IMyService, MyService>();
    services.AddControllersWithViews();
}
// Use the service in a controller
public class HomeController : Controller
{
    private readonly IMyService _myService;
    public HomeController(IMyService myService)
```

```

    {
        _myService = myService;
    }
    public IActionResult Index()
    {
        ViewBag.Message = _myService.GetMessage();
        return View();
    }
}

```

[Result]

When you navigate to the Index page, the message "Hello, Dependency Injection!" will be displayed.

Dependency injection (DI) is a core feature in ASP.NET Core, and it is implemented by default. DI helps in managing the dependencies of classes in a structured and controlled manner. Instead of classes instantiating their own dependencies, they receive these dependencies from an external source (the DI container), which is managed by the ASP.NET Core framework. In the example above, the service `MyService` implements the `IMyService` interface, and this service is registered in the `ConfigureServices` method of the `Startup` class. By using the `AddTransient` method, we tell the DI container to provide a new instance of `MyService` every time it is requested. When `HomeController` is created, ASP.NET Core automatically injects an instance of `IMyService` into it. This allows for loose coupling between the components of the application and makes the code easier to test and maintain.

[Trivia]

ASP.NET Core's dependency injection is built on the concept of Inversion of Control (IoC), where the control of creating objects and managing dependencies is inverted. Instead of the class controlling its dependencies, the framework takes care of it. This leads to cleaner, more modular, and testable code. ASP.NET Core's DI container is simple to use but can also be extended or replaced with more complex containers if needed.

3

Understanding Middleware in ASP.NET Core

ASP.NET Core utilizes middleware components to handle HTTP requests and responses. Middleware is a crucial part of the request processing pipeline, allowing developers to add functionality such as authentication, logging, and error handling.

The following code demonstrates how to configure middleware in the Startup class of an ASP.NET Core application.

[Code]

```
public class Startup
{
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        // Use developer exception page in development mode
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        // Use HTTPS redirection
        app.UseHttpsRedirection();
        // Use static files
        app.UseStaticFiles();
        // Custom middleware example
        app.Use(async (context, next) =>
        {
```

```

        // Custom logic before the next middleware
        Console.WriteLine("Request incoming: " +
context.Request.Path);
        await next.Invoke(); // Call the next middleware
        // Custom logic after the next middleware
        Console.WriteLine("Response outgoing: " +
context.Response.StatusCode);
    });
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
}

```

[Result]

When the application runs, it will log the incoming request path and the outgoing response status code to the console.

In this example, the Startup class contains a Configure method where middleware is added to the pipeline. The app.Use method is used to add custom middleware, which can execute code before and after the next middleware in the pipeline. The next.Invoke() call is essential as it allows the request to continue to the next middleware. The UseRouting and UseEndpoints methods set up the routing for the application, while UseStaticFiles enables serving static files.

[Trivia]

Middleware in ASP.NET Core is executed in the order it is added. This means that the first middleware added will be the first to handle the request and the last to handle the response. Understanding this order is crucial for debugging and optimizing the request processing pipeline.

4

Kestrel: The Default Web Server for ASP.NET Core

Kestrel is the default web server for ASP.NET Core applications. It is a lightweight, high-performance server designed to handle asynchronous requests efficiently, making it suitable for modern web applications.

The following command demonstrates how to run an ASP.NET Core application using the Kestrel web server.

[Code]

```
dotnet run
```

[Result]

The application will start, and you will see output similar to:
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.

When you run the command `dotnet run`, it builds and starts the application using Kestrel. By default, Kestrel listens on port 5000 for HTTP requests. You can access your application by navigating to `http://localhost:5000` in a web browser. Kestrel is designed to be fast and efficient, handling multiple requests simultaneously through asynchronous programming.

[Trivia]

Kestrel can be used in conjunction with a reverse proxy server like Nginx or IIS for production deployments. This setup enhances security and performance, as the reverse proxy can manage SSL termination, load balancing, and more. Additionally, Kestrel supports HTTP/2, which can improve the performance of web applications by allowing multiple requests and responses to be multiplexed over a single connection.

5

Razor Pages: A Simpler Alternative to MVC in ASP.NET Core

Razor Pages is a page-based framework that simplifies the process of building web pages in ASP.NET Core. Unlike the MVC pattern, Razor Pages encourages a more straightforward approach, keeping the code and HTML in close proximity, which is particularly helpful for those new to ASP.NET Core.

Below is a simple example of how to create a Razor Page that displays a greeting message. This example demonstrates the basic structure of a Razor Page and how to handle user input.

[Code]

```
// Pages/Index.cshtml.cs
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
public class IndexModel : PageModel
{
    [BindProperty]
    public string Name { get; set; }
    public string Message { get; private set; }
    public void OnPost()
    {
        if (!string.IsNullOrEmpty(Name))
        {
            Message = $"Hello, {Name}!";
        }
    }
}
html
<!-- Pages/Index.cshtml -->
@page
@model IndexModel
```

```
<form method="post">
  <label for="name">Enter your name:</label>
  <input type="text" id="name" name="Name" />
  <button type="submit">Submit</button>
</form>
@if (!string.IsNullOrEmpty(Model.Message))
{
  <p>@Model.Message</p>
}
```

[Result]

When a user enters their name in the input field and submits the form, the page will display a personalized greeting, e.g., "Hello, John!"

Razor Pages is part of the ASP.NET Core web framework and is designed to simplify the development of web applications. The key difference between Razor Pages and the traditional MVC framework is that Razor Pages follow a page-based routing model, which means that each page in the application is self-contained with its own view and logic. This is ideal for small to medium-sized applications where the overhead of separating controllers and views might be unnecessary. The `@page` directive at the top of `Index.cshtml` identifies the file as a Razor Page and handles incoming requests directly. The code-behind file, `Index.cshtml.cs`, contains the page's logic, such as processing form data and generating content for the view. Razor Pages also support model binding, as shown with the `[BindProperty]` attribute, which simplifies data transfer between the view and the code-behind file.

[Trivia]

Razor Pages were introduced in ASP.NET Core 2.0 to provide a simpler and more efficient way to build web applications compared to the MVC pattern. Razor Pages are ideal for scenarios where the page model fits naturally, like form submissions and content-focused pages.

6

Unified Web API and MVC Framework in ASP.NET Core

ASP.NET Core uses a unified framework that combines the strengths of both Web API and MVC patterns, allowing developers to build APIs and web applications seamlessly. This unification reduces redundancy and makes the framework more versatile and easier to learn.

Here's an example of how to create a basic API endpoint and an MVC view within the same ASP.NET Core project. This demonstrates the unified nature of the framework, where the same controllers can handle both web pages and API requests.

[Code]

```
// Controllers/HomeController.cs
using Microsoft.AspNetCore.Mvc;
namespace MyApp.Controllers
{
    public class HomeController : Controller
    {
        [HttpGet("/")]
        public IActionResult Index()
        {
            return View();
        }
        [HttpGet("/api/greet")]
        public IActionResult Greet(string name)
        {
            if (string.IsNullOrEmpty(name))
            {
                return BadRequest("Name cannot be empty");
            }
            return Ok(new { Message = $"Hello, {name}!" });
        }
    }
}
```



```

    }
}
html
<!-- Views/Home/Index.cshtml -->
@{
    ViewData["Title"] = "Home Page";
}
<h1>Welcome to ASP.NET Core!</h1>
<p>This is a simple web page served by the HomeController.</p>

```

[Result]

Accessing the root URL (/) will display the welcome page with the message "Welcome to ASP.NET Core!" Visiting /api/greet?name=John will return a JSON response: {"Message": "Hello, John!"}

The unification of Web API and MVC in ASP.NET Core means that developers do not need to learn different paradigms for building APIs and web applications. Both are handled by the same set of controllers, making the development process more consistent and streamlined. In the example, the HomeController is responsible for both rendering the HTML view and handling API requests. The Index action method returns a view, which is a typical MVC behavior, while the Greet action method returns a JSON response, demonstrating the Web API capability. This unified approach allows for easier management of both API and web application logic within the same codebase.

[Trivia]

The unified framework approach in ASP.NET Core allows for better reusability of code and more straightforward project architecture. By using the same controllers for both Web API and MVC, developers can avoid duplicating logic and maintain a more organized codebase. This feature also helps in maintaining consistency across different types of requests and responses in an application.

7

Understanding Configuration Hierarchy in ASP.NET Core

In ASP.NET Core, configuration is hierarchical and can be sourced from various places such as appsettings.json, environment variables, and command-line arguments. This allows for flexible and dynamic application settings management.

The following code demonstrates how to access configuration settings from different sources in an ASP.NET Core application.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using System;
var builder = WebApplication.CreateBuilder(args);
// Add configuration sources
builder.Configuration
    .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
    .AddEnvironmentVariables();
var app = builder.Build();
// Accessing configuration values
var mySetting = builder.Configuration["MySetting"];
var envSetting = builder.Configuration["EnvSetting"];
app.MapGet("/", () => $"MySetting: {mySetting}, EnvSetting:
{envSetting}");
app.Run();
```

[Result]

When you run the application, it will display the values of MySetting and EnvSetting from the configuration sources.

ASP.NET Core's configuration system is designed to be flexible and extensible. It supports multiple configuration providers that can read settings from different sources. The order in which these providers are added determines the precedence of the configuration values. For example, environment variables can override values specified in appsettings.json. This hierarchy allows developers to manage configurations across different environments (development, staging, production) seamlessly.

[Trivia]

The appsettings.json file is commonly used to store configuration settings in a structured JSON format. However, sensitive information such as connection strings or API keys should be stored in environment variables or a secure secrets management system to ensure security.

8

Caching Strategies in ASP.NET Core

ASP.NET Core provides multiple caching strategies including in-memory caching, distributed caching, and response caching to improve application performance by reducing the need to fetch or compute data repeatedly.

The following code demonstrates how to implement in-memory caching in an ASP.NET Core application.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using System;
var builder = WebApplication.CreateBuilder(args);
// Add in-memory caching service
builder.Services.AddMemoryCache();
var app = builder.Build();
app.MapGet("/", async (IMemoryCache cache) =>
{
    string cacheKey = "Time";
    if (!cache.TryGetValue(cacheKey, out string cachedTime))
    {
        // Set cache options
        var cacheEntryOptions = new MemoryCacheEntryOptions()
            .SetSlidingExpiration(TimeSpan.FromSeconds(60));
        // Save data in cache
        cachedTime = DateTime.Now.ToString();
        cache.Set(cacheKey, cachedTime, cacheEntryOptions);
    }
    return $"Cached Time: {cachedTime}";
});
app.Run();
```

[Result]

The application will display the cached time. If you refresh the page within 60 seconds, it will show the same time. After 60 seconds, the cache expires, and a new time will be cached and displayed.

Caching is a critical technique for enhancing application performance and scalability. In-memory caching stores data in the server's memory, making it fast but limited to a single server. Distributed caching, on the other hand, stores data in a central location accessible by multiple servers, making it suitable for cloud or multi-server environments. Response caching caches HTTP responses to reduce server load and improve response times for repeated requests.

[Trivia]

In-memory caching is ideal for small to medium-sized applications running on a single server. However, for large-scale applications with multiple servers, distributed caching solutions like Redis or SQL Server are recommended to ensure data consistency and availability across all servers.

9

Entity Framework Core for Database Interactions

Entity Framework Core (EF Core) is a modern Object-Relational Mapper (ORM) for .NET that enables developers to interact with databases using .NET objects.

Below is a simple example of how to use EF Core to perform basic CRUD (Create, Read, Update, Delete) operations in a database. This example demonstrates setting up a DbContext and interacting with an in-memory database.

[Code]

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
// Define a model class
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
// Define a DbContext
public class AppDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }
    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        optionsBuilder.UseInMemoryDatabase("TestDb");
    }
}
```

```

class Program
{
    static void Main()
    {
        using (var context = new AppDbContext())
        {
            // Create
            var product = new Product { Name = "Laptop", Price =
1200.00M };
            context.Products.Add(product);
            context.SaveChanges();
            // Read
            var products = context.Products.ToList();
            Console.WriteLine("Products in database:");
            foreach (var p in products)
            {
                Console.WriteLine($"- {p.Name} costs {p.Price}");
            }
            // Update
            product.Price = 1100.00M;
            context.SaveChanges();
            Console.WriteLine($"Updated Price:
{context.Products.First().Price}");
            // Delete
            context.Products.Remove(product);
            context.SaveChanges();
            Console.WriteLine($"Number of products after deletion:
{context.Products.Count()}");
        }
    }
}

```

[Result]

Products in database:
 - Laptop costs 1200.00
 Updated Price: 1100.00
 Number of products after deletion: 0

Entity Framework Core is a powerful ORM that simplifies data access by allowing developers to work with data using .NET objects rather than SQL queries. It supports LINQ queries, change tracking, updates, and schema migrations. In the example above, an in-memory database is used for simplicity, making it ideal for testing and prototyping. The `DbContext` class is the primary class responsible for interacting with the database, and `DbSet<T>` represents a collection of entities of a specific type.

[Trivia]

EF Core supports multiple database providers, including SQL Server, SQLite, PostgreSQL, and MySQL. It is highly extensible and can be configured to work with custom conventions and behaviors. Additionally, EF Core offers features like lazy loading, eager loading, and explicit loading to manage how related data is retrieved.

10

ASP.NET Core Identity for User Authentication

ASP.NET Core Identity is a membership system that provides login functionality, user management, and security features for ASP.NET Core applications.

The following example demonstrates how to set up ASP.NET Core Identity in a web application. It includes basic user registration and login functionality using Identity's built-in services.

[Code]

```
// Startup.cs
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<ApplicationDbContext>(options =>
            options.UseInMemoryDatabase("IdentityDb"));
        services.AddIdentity<IdentityUser, IdentityRole>()
            .AddEntityFrameworkStores<ApplicationDbContext>()
            .AddDefaultTokenProviders();
        services.AddControllersWithViews();
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Home/Error");
        }
    }
}
```

```

        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthentication();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
}
// AccountController.cs
public class AccountController : Controller
{
    private readonly UserManager<IdentityUser> _userManager;
    private readonly SignInManager<IdentityUser> _signInManager;
    public AccountController(UserManager<IdentityUser> userManager,
SignInManager<IdentityUser> signInManager)
    {
        _userManager = userManager;
        _signInManager = signInManager;
    }
    [HttpPost]
    public async Task<IActionResult> Register(string email, string
password)
    {
        var user = new IdentityUser { UserName = email, Email = email };
        var result = await _userManager.CreateAsync(user, password);
        if (result.Succeeded)
        {
            await _signInManager.SignInAsync(user, isPersistent: false);
            return RedirectToAction("Index", "Home");
        }
        return View();
    }
    [HttpPost]

```

```
public async Task<IActionResult> Login(string email, string
password)
{
    var result = await _signInManager.PasswordSignInAsync(email,
password, isPersistent: false, lockoutOnFailure: false);
    if (result.Succeeded)
    {
        return RedirectToAction("Index", "Home");
    }
    return View();
}
```

[Result]

The application allows users to register and log in. Successful registration or login redirects the user to the home page.

ASP.NET Core Identity provides a comprehensive system for managing user authentication and authorization. It includes features like password hashing, account lockout, two-factor authentication, and external login providers (e.g., Google, Facebook). The UserManager and SignInManager classes are central to handling user accounts and sign-in processes. Identity can be customized extensively to fit the specific requirements of an application.

[Trivia]

ASP.NET Core Identity is built on top of Entity Framework Core, allowing it to store user information in a database. It can be configured to use different storage mechanisms, such as NoSQL databases or custom stores. Identity also supports role-based authorization, enabling developers to restrict access to certain parts of an application based on user roles.

11

Understanding Tag Helpers in ASP.NET Core

Tag Helpers in ASP.NET Core allow developers to use server-side code to generate and render HTML elements, enhancing the functionality and maintainability of Razor views.

Tag Helpers are a feature in ASP.NET Core that enable developers to write server-side logic to manipulate HTML elements in Razor views. They provide a cleaner and more intuitive way to work with HTML, as opposed to traditional HTML helpers.

[Code]

```
// Example of a custom Tag Helper in ASP.NET Core
using Microsoft.AspNetCore.Razor.TagHelpers;
public class EmailTagHelper : TagHelper
{
    public string Address { get; set; }
    public string Domain { get; set; }
    public override void Process(TagHelperContext context,
TagHelperOutput output)
    {
        output.TagName = "a"; // Replaces <email> with <a> tag
        var address = $"{Address}@{Domain}";
        output.Attributes.SetAttribute("href", $"mailto:{address}");
        output.Content.SetContent(address);
    }
}
// Usage in a Razor view
<email address="info" domain="example.com"></email>
```

[Result]

```
<a href="mailto:info@example.com">info@example.com</a>
```

Tag Helpers are processed on the server side and allow you to create reusable components that encapsulate both behavior and rendering logic. They are defined by creating a class that inherits from `TagHelper` and overriding the `Process` method. This method allows you to manipulate the output of the HTML tag. Tag Helpers are automatically discovered by ASP.NET Core if they are in the same assembly as your application or referenced in your project.

[Trivia]

Tag Helpers are case-insensitive and can be applied to any HTML element. They are a powerful feature that can be used to enforce consistency across your views and reduce code duplication by encapsulating common patterns.

12

Logging in ASP.NET Core with Built-in Support

ASP.NET Core provides built-in support for logging, allowing developers to log information to various outputs, known as sinks, using different providers.

ASP.NET Core's logging framework is flexible and allows developers to log messages to different outputs, such as the console, files, or third-party services. This is achieved through logging providers, which can be configured in the application's startup process.

[Code]

```
// Example of configuring logging in ASP.NET Core
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddLogging(config =>
        {
            config.AddConsole(); // Adds console logging
            config.AddDebug(); // Adds debug logging
        });
    }
    public void Configure(IApplicationBuilder app, ILogger<Startup>
logger)
    {
        app.Run(async (context) =>
        {
            logger.LogInformation("Handling request.");
            await context.Response.WriteAsync("Hello, world!");
        });
    }
}
```

```
        logger.LogInformation("Finished handling request.");
    });
}
```

[Result]

Logs will be output to the console and debug window, showing messages like:

```
info: Startup[0]
      Handling request.
info: Startup[0]
      Finished handling request.
```

ASP.NET Core's logging framework is built around the ILogger interface, which provides methods for logging messages at different severity levels (e.g., Information, Warning, Error). The logging system is highly extensible, allowing you to add custom providers or configure existing ones to suit your needs. Providers like Console, Debug, EventSource, and others can be easily added to capture logs in various environments.

[Trivia]

The logging framework in ASP.NET Core is designed to be non-blocking and efficient, using a structured logging approach that allows you to include additional context with your log messages. This makes it easier to filter and analyze logs in production environments.

13

Health Checks in ASP.NET Core for Application Monitoring

ASP.NET Core provides a built-in feature called Health Checks to monitor the health of your application. It allows you to verify the status of various components like databases, external services, and application endpoints.

Health Checks in ASP.NET Core are used to determine if an application and its dependencies are functioning correctly. This is crucial for maintaining the reliability and availability of your application.

[Code]

```
// 1. Install the necessary NuGet package:
// dotnet add package Microsoft.AspNetCore.Diagnostics.HealthChecks
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Diagnostics.HealthChecks;
using Microsoft.Extensions.Hosting;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // 2. Add Health Checks to the services collection
        services.AddHealthChecks()
            .AddCheck("SampleHealthCheck", () =>
                HealthCheckResult.Healthy("The application is healthy!"));
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {

```



```
        app.UseDeveloperExceptionPage();
    }
    // 3. Enable Health Checks endpoint
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapHealthChecks("/health");
    });
}
```

[Result]

When you run the application and navigate to /health, you'll see the following output:

Healthy

This indicates that the application is running as expected.

Health Checks are particularly useful in environments where applications are running in containers or distributed systems. By exposing a /health endpoint, you can allow orchestrators like Kubernetes or load balancers to automatically check the health of your application and take action if it's not functioning correctly. This could include restarting the container or redirecting traffic to a healthy instance. To customize health checks, you can add more checks (e.g., checking database connections or external APIs), and you can configure them to return Unhealthy or Degraded statuses depending on specific conditions. Health Checks also support filtering and reporting through libraries like `AspNetCore.HealthChecks.UI`.

[Trivia]

ASP.NET Core Health Checks can be integrated with third-party monitoring tools such as Prometheus or Datadog, allowing for centralized monitoring and alerting across multiple applications and services.

14

Environment-Specific Behavior in ASP.NET Core Hosting Environment

The Hosting Environment in ASP.NET Core allows developers to specify behavior based on the environment in which the application is running. This is crucial for managing different configurations and behaviors for Development, Staging, and Production environments.

The Hosting Environment feature in ASP.NET Core enables your application to behave differently depending on the environment, such as loading different configurations or enabling debugging tools in development but disabling them in production.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
public class Startup
{
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            // 1. Development-specific configuration
            app.UseDeveloperExceptionPage(); // Detailed error pages in
development
        }
        else if (env.IsStaging())
        {
            // 2. Staging-specific configuration
            app.UseExceptionHandler("/Error"); // Use a generic error page
in staging
        }
    }
}
```

```

else
{
    // 3. Production-specific configuration
    app.UseExceptionHandler("/Error");
    app.UseHsts(); // Enforce HTTPS in production
}
app.UseHttpsRedirection();
app.UseRouting();
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync($"Environment:
{env.EnvironmentName}");
    });
});
}
}

```

[Result]

When you run the application in different environments, the output on the main page will reflect the current environment:Environment:

Development

Ormakefile

Environment: Production

The behavior of the application (e.g., error handling, HTTPS enforcement) will also change based on the environment.

ASP.NET Core determines the environment through the `ASPNETCORE_ENVIRONMENT` environment variable. By default, this variable can be set to Development, Staging, or Production. The environment variable can be set in various ways, such as in the launch settings file (launchSettings.json), through the command line, or directly on the hosting server. This feature allows you to create flexible and secure applications by ensuring that debugging tools are only available in development, while performance optimizations and security features are enforced in production. Additionally, environment-specific configuration

files (`appsettings.Development.json`, `appsettings.Production.json`) allow you to customize settings like connection strings and service URLs depending on the environment.

[Trivia]

It's recommended to avoid hardcoding environment-specific logic in your application code. Instead, use dependency injection and environment-specific configuration files to manage differences between environments. This approach keeps your codebase clean and maintainable.

15

Hosting Options for ASP.NET Core Applications

ASP.NET Core applications can be hosted using different web servers such as IIS, Nginx, Apache, or as standalone processes. This flexibility allows developers to choose the best hosting environment for their needs.

In this section, we'll explore how to host an ASP.NET Core application using IIS and as a standalone process. We'll provide examples of configuration and deployment.

[Code]

```
// Example of hosting ASP.NET Core in IIS
// 1. Install the .NET Core Hosting Bundle on the server.
// 2. Publish your application using the command:
//    dotnet publish --configuration Release
// 3. Configure IIS to point to the published folder.
// 4. Ensure the application pool is set to "No Managed Code".
// Example of hosting ASP.NET Core as a standalone process
// 1. Publish your application using the command:
//    dotnet publish --configuration Release
// 2. Navigate to the publish folder and run:
//    dotnet YourApp.dll
```

[Result]

When hosted in IIS, the application will be accessible through the configured domain or IP address. As a standalone process, the application will be accessible at the specified port (e.g., <http://localhost:5000>).

Hosting in IIS provides integration with Windows authentication and other IIS features. It acts as a reverse proxy, forwarding requests to the

Kestrel server running your application. Standalone hosting with Kestrel is lightweight and cross-platform, suitable for Linux and macOS environments. When using Nginx or Apache, they typically act as reverse proxies to Kestrel, offering additional features like load balancing and SSL termination.

[Trivia]

Kestrel is the default web server included with ASP.NET Core. It is designed to be fast and efficient, but for production scenarios, it is often used behind a reverse proxy like IIS, Nginx, or Apache for enhanced security and performance.

16

Understanding Minimal APIs in ASP.NET Core

Minimal APIs in ASP.NET Core 6.0 and later offer a simplified way to create HTTP APIs with minimal setup, making it easier and faster to build microservices.

We'll demonstrate how to create a Minimal API in ASP.NET Core. This approach reduces boilerplate code and is ideal for small applications or microservices.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Hosting;
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();
app.MapGet("/", () => "Hello, World!");
app.Run();
```

[Result]

When running the above code, navigating to <http://localhost:5000> will display "Hello, World!" in the browser.

Minimal APIs streamline the process of creating HTTP endpoints by leveraging top-level statements and a simplified hosting model. This approach reduces the need for controllers and attributes, focusing on concise and readable code. It is particularly beneficial for developers familiar with Node.js or Flask, who prefer a more functional programming style.

[Trivia]

Minimal APIs support dependency injection, middleware, and routing, just like traditional ASP.NET Core applications. They are part of the effort to make .NET more accessible to developers who prefer minimalistic and straightforward coding approaches.

Global Error Handling in ASP.NET Core with UseExceptionHandler Middleware

In ASP.NET Core, you can implement global error handling by using the `UseExceptionHandler` middleware. This middleware allows you to capture exceptions that occur during request processing and direct them to a specific error handling path, ensuring that all unhandled exceptions are managed in a consistent way.

To implement global error handling, you typically configure the `UseExceptionHandler` middleware in the `Startup` class of your ASP.NET Core application. This middleware can be set up to route the user to an error page or return a specific error response in case of an exception.

[Code]

```
public class Startup
{
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            // Global error handler configuration
            app.UseExceptionHandler("/Home/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();
    }
}
```

```

app.UseRouting();
app.UseAuthorization();
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
}
// Error Controller
public class HomeController : Controller
{
    [Route("Home/Error")]
    public IActionResult Error()
    {
        return View();
    }
}

```

[Result]

When an unhandled exception occurs in the application, the user is redirected to the /Home/Error route, where they will see the error page defined by the Error action in the HomeController.

The `UseExceptionHandler` middleware is an essential component for managing unhandled exceptions in a production environment. When an exception occurs, rather than exposing potentially sensitive error details to the user, the middleware reroutes the request to a specific error-handling endpoint. This improves the security and user experience of your application. In the example, during development, the `UseDeveloperExceptionPage` is used, which provides detailed exception information, useful for debugging. In a production environment, this should be replaced with `UseExceptionHandler` to prevent exposing internal details.

[Trivia]

It's important to note that `UseExceptionHandler` does not handle exceptions thrown outside of the request processing pipeline, such as those that occur during application startup. For such cases, you may need to implement additional error handling logic or configure the host to manage startup errors appropriately.

18

High-Performance Remote Procedure Calls with gRPC in ASP.NET Core

ASP.NET Core supports gRPC, a high-performance, open-source framework for making remote procedure calls (RPC). gRPC is ideal for scenarios where efficiency and low latency are critical, and it is fully supported on cross-platform environments.

To use gRPC in an ASP.NET Core application, you need to define a service in a .proto file, implement the service in your application, and configure the necessary middleware to handle gRPC requests.

[Code]

```
// Define the gRPC service in a .proto file
syntax = "proto3";
option csharp_namespace = "GrpcServiceExample";
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply);
}
message HelloRequest {
  string name = 1;
}
message HelloReply {
  string message = 1;
}
csharp
// Implement the service in your ASP.NET Core application
public class GreeterService : Greeter.GreeterBase
{
    public override Task<HelloReply> SayHello(HelloRequest request,
ServerCallContext context)
    {
        return Task.FromResult(new HelloReply
        {
```

```

        Message = "Hello " + request.Name
    });
}
}
// Configure gRPC middleware in Startup
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddGrpc();
    }
    public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGrpcService<GreeterService>();
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Communication with
gRPC endpoints must be made through a gRPC client.");
            });
        });
    }
}

```

[Result]

When the gRPC server is running, clients can make requests to the SayHello method. The server will respond with a message such as "Hello [Name]", where [Name] is the name provided in the request.

gRPC is highly efficient because it uses Protocol Buffers (protobuf) as its interface definition language. Protobuf is a binary format that is much smaller and faster to serialize and deserialize than JSON or XML, which are commonly used in REST APIs. This makes gRPC suitable for high-performance applications, such as microservices architectures, real-time communication, and when working with limited bandwidth

environments.gRPC in ASP.NET Core fully supports features like streaming, bi-directional communication, and built-in authentication. It is also cross-platform, which means you can implement gRPC services in a .NET environment and consume them from clients written in other languages, such as Java, Python, or Go.

[Trivia]

gRPC requires HTTP/2, which provides several performance advantages over HTTP/1.x, including lower latency and more efficient use of network resources. However, it also means that gRPC is not natively supported in some environments, such as web browsers. To overcome this, gRPC-Web can be used as a proxy to enable gRPC communication in browser-based applications.

19

Understanding appsettings.json in ASP.NET Core

In ASP.NET Core, the appsettings.json file is used to store configuration settings for your application. This file allows you to manage application settings in a structured and easily editable format.

The following code demonstrates how to read configuration settings from the appsettings.json file.

[Code]

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((context, config) =>
            {
                // Accessing configuration settings
```

```
        var settings = config.Build();
        var logLevel = settings["Logging:LogLevel:Default"];
        Console.WriteLine($"Default Log Level: {logLevel}");
    })
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
}
```

[Result]

Default Log Level: Information

The appsettings.json file is structured in JSON format, which makes it easy to read and modify. In this example, we define logging levels for different components of the application. The CreateHostBuilder method builds the application host and allows us to access the configuration settings. The line `var logLevel = settings["Logging:LogLevel:Default"];` retrieves the default log level from the configuration file and prints it to the console. This approach helps in managing different environments by creating separate configuration files like `appsettings.Development.json` or `appsettings.Production.json`.

[Trivia]

ASP.NET Core supports hierarchical configuration, allowing you to nest settings within sections.

You can also use environment variables and command-line arguments to override settings in `appsettings.json`.

The `IConfiguration` interface is key for accessing configuration settings throughout your application.

Introduction to Action Filters in ASP.NET Core

Action Filters in ASP.NET Core are attributes that allow you to run code before or after an action method executes. They are useful for cross-cutting concerns such as logging, authorization, and caching.

The following code example illustrates how to create a custom action filter that logs the execution time of an action method.

[Code]

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System.Diagnostics;
public class LogExecutionTimeAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext
context)
    {
        // Start timing
        context.HttpContext.Items["StartTime"] =
Stopwatch.GetTimestamp();
    }
    public override void OnActionExecuted(ActionExecutedContext
context)
    {
        // Calculate execution time
        var startTime = (long)context.HttpContext.Items["StartTime"];
        var executionTime = Stopwatch.GetTimestamp() - startTime;
        var elapsedMilliseconds = (executionTime /
(double)Stopwatch.Frequency) * 1000;
        Console.WriteLine($"Action executed in {elapsedMilliseconds}
ms");
    }
}
```

```

}
[ApiController]
[Route("[controller]")]
public class SampleController : ControllerBase
{
    [HttpGet]
    [LogExecutionTime]
    public IActionResult Get()
    {
        // Simulate some work
        System.Threading.Thread.Sleep(500);
        return Ok("Hello, World!");
    }
}

```

[Result]

Action executed in 500.1234 ms (the exact time may vary)

In this example, we define a custom action filter `LogExecutionTimeAttribute` that derives from `ActionFilterAttribute`. The `OnActionExecuting` method is called before the action method starts, where we record the current timestamp. The `OnActionExecuted` method is called after the action method completes, allowing us to calculate and log the execution time. The action method `Get` simulates a delay using `Thread.Sleep`, and when the action is executed, the filter logs how long it took to complete.

[Trivia]

Action filters can be applied globally, at the controller level, or on individual action methods.

ASP.NET Core provides several built-in action filters such as `Authorize`, `ServiceFilter`, and `TypeFilter`.

Filters can also be used to modify the result of an action method or handle exceptions.

Data Protection in ASP.NET Core: Encrypting and Decrypting Data

Data protection in ASP.NET Core is a crucial feature that helps developers secure sensitive data by encrypting and decrypting it. This is particularly important for maintaining user privacy and ensuring data integrity.

The following code demonstrates how to set up data protection services and use them to encrypt and decrypt a simple string.

[Code]

```
using Microsoft.AspNetCore.DataProtection;
using Microsoft.Extensions.DependencyInjection;
public class Program
{
    public static void Main(string[] args)
    {
        // Set up the service collection
        var services = new ServiceCollection();
        services.AddDataProtection();
        // Build the service provider
        var serviceProvider = services.BuildServiceProvider();
        // Create a data protector
        var protector =
serviceProvider.GetDataProtectionProvider().CreateProtector("MyPurpo
se");
        // Encrypt a string
        string originalData = "Sensitive Data";
        string encryptedData = protector.Protect(originalData);
        Console.WriteLine($"Encrypted: {encryptedData}");
        // Decrypt the string
        string decryptedData = protector.Unprotect(encryptedData);
        Console.WriteLine($"Decrypted: {decryptedData}");
    }
}
```

```
}  
}
```

[Result]

Encrypted: [Encrypted_String]
Decrypted: Sensitive Data

In the code above, we first set up a service collection and add the data protection services. We then create a data protector with a specific purpose (in this case, "MyPurpose"). The Protect method is used to encrypt the original data, while the Unprotect method is used to decrypt it back to its original form. The encrypted string will look like a random sequence of characters, while the decrypted string will return to "Sensitive Data".

[Trivia]

Data protection in ASP.NET Core uses a key management system to store encryption keys securely. By default, these keys are stored in memory, but you can also configure it to store keys in a file or a database for persistence across application restarts. This is essential for applications that need to maintain data security over time.

22

SignalR in ASP.NET Core: Real-Time Web Functionality

SignalR is a library in ASP.NET Core that enables real-time web functionality, allowing server-side code to push content to connected clients instantly. This is particularly useful for applications that require live updates, such as chat applications or real-time dashboards.

The following code illustrates how to set up a simple SignalR hub and connect a client to receive messages in real-time.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.SignalR;
using Microsoft.Extensions.DependencyInjection;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSignalR();
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapHub<ChatHub>("/chatHub");
        });
    }
}
public class ChatHub : Hub
{
}
```

```
public async Task SendMessage(string user, string message)
{
    await Clients.All.SendAsync("ReceiveMessage", user, message);
}
```

[Result]

No direct output as this is a server-side code. The client must connect to the hub to receive messages.

In this example, we first configure the SignalR service in the `ConfigureServices` method. The `ChatHub` class inherits from `Hub`, which is the core component of SignalR that manages connections and messaging. The `SendMessage` method allows clients to send messages to all connected users. The actual output will depend on the client-side implementation that connects to this hub and listens for messages.

[Trivia]

SignalR supports various transport methods, including WebSockets, Server-Sent Events, and Long Polling. WebSockets provide the best performance, but SignalR automatically falls back to other methods if WebSockets are not available. Additionally, SignalR can scale out to multiple servers using backplane technologies like Redis or Azure SignalR Service, making it suitable for large applications.

23

Understanding Model Binding in ASP.NET Core

Model Binding in ASP.NET Core maps data from HTTP requests to action method parameters, making it easier to work with user input.

Let's explore how ASP.NET Core automatically maps incoming request data to parameters in controller actions using Model Binding.

[Code]

```
// SampleController.cs
using Microsoft.AspNetCore.Mvc;
namespace MyApp.Controllers
{
    public class SampleController : Controller
    {
        // This action method uses model binding to map query string
        parameters to method parameters
        [HttpGet("api/sample")]
        public IActionResult GetSampleData(int id, string name)
        {
            return Ok(new { Id = id, Name = name });
        }
    }
}
```

[Result]

When you access the URL `/api/sample?id=1&name=John`, the response will be:

```
{
  "Id": 1,
  "Name": "John"
}
```

```
}
```

Model Binding in ASP.NET Core is a powerful feature that simplifies the process of handling input data. It supports binding from various sources such as query strings, form data, route data, and more. The framework automatically converts these inputs into the appropriate data types for your action method parameters. If the binding fails (e.g., due to type mismatches), the framework can handle errors gracefully, often by returning a model state error.

[Trivia]

Model Binding can also work with complex types, not just simple data types. For instance, if you have a class `Person` with properties `Id` and `Name`, you can use it directly as a parameter in your action method, and ASP.NET Core will populate its properties from the request data.

24

Creating Custom Middleware in ASP.NET Core

Custom middleware in ASP.NET Core allows developers to handle specific requests or add processing steps to the request pipeline.

Learn how to create a custom middleware component to log request details in an ASP.NET Core application.

[Code]

```
// RequestLoggingMiddleware.cs
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
public class RequestLoggingMiddleware
{
    private readonly RequestDelegate _next;
    public RequestLoggingMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    public async Task InvokeAsync(HttpContext context)
    {
        // Log request details
        Console.WriteLine($"Request: {context.Request.Method}
{context.Request.Path}");
        // Call the next middleware in the pipeline
        await _next(context);
    }
}
// Startup.cs
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {

```

```
app.UseMiddleware<RequestLoggingMiddleware>();
app.UseRouting();
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
});
}
```

[Result]

When a request is made, the console will display:
Request: GET /api/sample

Middleware in ASP.NET Core is a way to compose an application's request pipeline using a series of request delegates. Each middleware component can perform operations before and after the next component in the pipeline. Custom middleware is defined by creating a class with an `InvokeAsync` method that takes an `HttpContext` parameter. This method can perform any logic you need, such as logging, authentication, or modifying the request or response.

[Trivia]

Middleware components are executed in the order they are added to the request pipeline. This order is crucial because it determines how requests are processed and how responses are generated. If you place a middleware component that short-circuits the pipeline (e.g., returning a response without calling the next delegate), subsequent middleware will not execute.

Customizable Routing in ASP.NET Core

ASP.NET Core's routing system allows developers to define routes using attributes or conventions, providing flexibility in how requests are handled.

In ASP.NET Core, you can define routes using attribute routing or convention-based routing. Attribute routing is defined directly on the action methods, while convention-based routing is defined in the Startup.cs file. Here is an example demonstrating both methods.

[Code]

```
// Attribute-based routing example
using Microsoft.AspNetCore.Mvc;
namespace MyApp.Controllers
{
    [Route("api/[controller]")]
    public class ProductsController : Controller
    {
        // GET api/products
        [HttpGet]
        public IActionResult GetAll()
        {
            return Ok(new string[] { "Product1", "Product2" });
        }
        // GET api/products/5
        [HttpGet("{id}")]
        public IActionResult GetById(int id)
        {
            return Ok($"Product {id}");
        }
    }
}
// Convention-based routing example in Startup.cs
using Microsoft.AspNetCore.Builder;
```

```

using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllerRoute(
                name: "default",
                pattern: "{controller=Home}/{action=Index}/{id?}");
        });
    }
}

```

[Result]

When accessing `api/products`, the response will be `["Product1", "Product2"]`. Accessing `api/products/5` will return Product 5.

Attribute routing allows you to specify routes directly on your controller actions, making it easy to see the route configuration in one place. This is especially useful for APIs where routes are closely tied to the actions they map to. Convention-based routing, on the other hand, is defined in the `Startup.cs` file and provides a centralized way to define routes, which can be beneficial for larger applications where you want consistent routing patterns.

[Trivia]

ASP.NET Core's routing system is built on top of the middleware pipeline, allowing for powerful and flexible routing capabilities. It supports route constraints, custom route handlers, and can be extended to support custom routing logic.

Building Web UIs with Blazor and C#

Blazor enables developers to create interactive web user interfaces using C# instead of JavaScript, leveraging the power of .NET.

Blazor is a framework for building interactive web applications using C#. It allows developers to write client-side logic in C# and run it in the browser using WebAssembly. Here is a simple example of a Blazor component that demonstrates this capability.

[Code]

```
@page "/counter"
<h3>Counter</h3>
<p>Current count: @currentCount</p>
<button class="btn btn-primary" @onclick="IncrementCount">Click
me</button>
@code {
    private int currentCount = 0;
    private void IncrementCount()
    {
        currentCount++;
    }
}
```

[Result]

When you navigate to the /counter page and click the "Click me" button, the counter will increment, displaying the updated count.

Blazor provides a component-based architecture similar to React or Angular. Components are reusable UI elements that encapsulate rendering logic and behavior. Blazor supports both server-side and client-side hosting models, allowing for flexibility in deployment. With Blazor,

you can share code between the client and server, use .NET libraries, and benefit from the strong typing and tooling support of C#.

[Trivia]

Blazor WebAssembly runs .NET code directly in the browser through WebAssembly, a binary instruction format that allows for near-native performance. This means you can leverage the full power of .NET and C# in the browser without relying on JavaScript.

Configuring the Request Pipeline with IApplicationBuilder

The `IApplicationBuilder` interface in ASP.NET Core is essential for configuring the application's request pipeline. It defines the sequence of middleware components that process HTTP requests, making it crucial for handling requests and responses efficiently.

In ASP.NET Core, the `IApplicationBuilder` interface is used within the `Startup.Configure` method to build the request pipeline. This example demonstrates how to add middleware components to the pipeline using `Use`, `UseRouting`, and `UseEndpoints` methods.

[Code]

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        // Adding a simple middleware that writes "Hello World!" to the
        response
        app.Use(async (context, next) =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
        // Adding routing middleware
        app.UseRouting();
        // Adding endpoint middleware
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapGet("/", async context =>
            {
                await context.Response.WriteAsync("Welcome to the Home
                Page!");
            });
        });
    }
}
```

```
    });  
  }  
}
```

[Result]

When accessing the root URL (/), the browser displays the message "Welcome to the Home Page!". If the root URL is not accessed, the response will be "Hello World!".

The `IApplicationBuilder` interface allows you to define the order in which middleware components are invoked. Middleware is any piece of code that processes a request, and its order in the pipeline can significantly affect how requests are handled. Middleware components can perform actions such as logging, authentication, and response compression. The `UseRouting` middleware enables routing in the application, while `UseEndpoints` maps requests to endpoints, such as controllers or Razor Pages. Understanding the flow of middleware is crucial because each middleware component can either pass control to the next component in the pipeline or terminate the request by generating a response.

[Trivia]

Middleware components are executed in the order they are added to the pipeline, which means the sequence of `app.Use` calls matters. The `IApplicationBuilder` interface provides methods like `Use`, `Run`, and `Map`, each serving different purposes in controlling the request pipeline. A common practice is to use extension methods on `IApplicationBuilder` to encapsulate middleware configuration logic, making the code more modular and readable.

Understanding the IHostingEnvironment Interface

The IHostingEnvironment interface provides information about the environment in which an ASP.NET Core application is running, such as Development, Staging, or Production. This interface is useful for configuring services and middleware differently based on the environment.

The IHostingEnvironment interface is typically used in the Startup.Configure or Startup.ConfigureServices methods to apply environment-specific logic. This example shows how to use IHostingEnvironment to execute code only in the development environment.

[Code]

```
public class Startup
{
    private readonly IHostingEnvironment _env;
    public Startup(IHostingEnvironment env)
    {
        _env = env;
    }
    public void Configure(IApplicationBuilder app)
    {
        if (_env.IsDevelopment())
        {
            // Use the Developer Exception Page only in Development
environment
            app.UseDeveloperExceptionPage();
        }
        else
        {
            // Handle errors differently in Production

```

```

        app.UseExceptionHandler("/Home/Error");
    }
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Environment: " +
            _env.EnvironmentName);
        });
    });
}
}

```

[Result]

When the application runs in the Development environment, it uses the Developer Exception Page for detailed error information. In other environments, it redirects to the /Home/Error page. The root URL displays the current environment name.

The `IHostingEnvironment` interface is a critical component in ASP.NET Core applications, enabling developers to tailor the application's behavior based on the environment. Common environments include Development, Staging, and Production, each requiring different configurations. For instance, in Development, you might want detailed error messages and additional logging, whereas in Production, you would prefer streamlined logging and user-friendly error pages. The `IsDevelopment()`, `IsStaging()`, and `IsProduction()` methods help in writing conditional code based on the environment. The environment name can be set through various means, including environment variables, configuration files, or even command-line arguments.

[Trivia]

The `IHostingEnvironment` interface has been superseded by `IWebHostEnvironment` in ASP.NET Core 3.0 and later, but the core concepts remain the same. Environment-specific settings are often stored in `appsettings.{Environment}.json` files, allowing you to override configurations based on the environment. Using the environment name in

conditional logic helps ensure that sensitive debugging information is not exposed in Production, improving the security of your application.

Default HTTPS Support in ASP.NET Core

ASP.NET Core supports HTTPS by default and strongly encourages its use in production environments to ensure secure data transmission.

ASP.NET Core applications are configured to use HTTPS by default, which helps protect data integrity and confidentiality. Below is an example of how to enforce HTTPS in an ASP.NET Core application.

[Code]

```
// In the Startup.cs file, configure the application to use HTTPS
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts(); // Enforces the use of HTTPS
        }
        app.UseHttpsRedirection(); // Redirects HTTP requests to HTTPS
        app.UseStaticFiles();
        app.UseRouting();
        app.UseAuthorization();
    }
}
```

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
});
}
```

[Result]

When you run the application, any HTTP requests will be automatically redirected to HTTPS.

HTTPS (Hypertext Transfer Protocol Secure) is an extension of HTTP. It is used for secure communication over a computer network and is widely used on the Internet. In ASP.NET Core, HTTPS is enabled by default to protect sensitive data from being intercepted by attackers. The `UseHttpsRedirection` middleware ensures that all HTTP requests are redirected to HTTPS, while `UseHsts` adds the HTTP Strict Transport Security (HSTS) header to responses, which instructs browsers to only interact with the server using HTTPS.

[Trivia]

ASP.NET Core's HTTPS support is part of its commitment to security. The framework provides built-in support for configuring certificates and handling HTTPS redirection, making it easier for developers to secure their applications without extensive manual configuration.

Deployment Options in ASP.NET Core

ASP.NET Core applications can be deployed as either self-contained or framework-dependent, offering flexibility in deployment scenarios.

ASP.NET Core provides two main deployment modes: self-contained and framework-dependent. This flexibility allows developers to choose how their applications are packaged and run on different environments.

[Code]

```
# Framework-dependent deployment
dotnet publish -c Release
# Self-contained deployment for Windows
dotnet publish -c Release -r win-x64 --self-contained true
# Self-contained deployment for Linux
dotnet publish -c Release -r linux-x64 --self-contained true
```

[Result]

The result of these commands is a set of files in the publish directory. For framework-dependent deployments, the application relies on the .NET runtime installed on the host machine. For self-contained deployments, all necessary runtime files are included, allowing the application to run on a machine without any prior .NET installation.

In a framework-dependent deployment, the application relies on the presence of a shared .NET runtime on the host machine. This approach results in a smaller package size and is suitable for environments where the runtime is already installed. In contrast, a self-contained deployment includes the .NET runtime and all necessary libraries within the package, making it larger but ensuring that the application can run independently of the host's environment. This is useful for scenarios where you cannot guarantee the presence of the .NET runtime on the target machine.

[Trivia]

Self-contained deployments are particularly useful for distributing applications to environments where you do not have control over the installed software, such as customer machines or cloud environments. They ensure that your application behaves consistently, regardless of the underlying system configuration.

Built-in Support for Data Formats in ASP.NET Core APIs

ASP.NET Core provides built-in support for handling various data formats such as JSON and XML in APIs, making it easier to create web services that can communicate with different clients.

In ASP.NET Core, you can easily configure your API to handle JSON and XML data formats. This is crucial for building RESTful services that need to interact with different types of clients.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
namespace DataFormatExample
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            // Add controllers with JSON and XML formatters
            services.AddControllers()
                .AddXmlSerializerFormatters(); // Enables XML support
        }
        public void Configure(IApplicationBuilder app,
            IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
        }
    }
}
```



```

        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
[ApiController]
[Route("[controller]")]
public class SampleController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        var data = new { Message = "Hello, World!" };
        return Ok(data); // Returns JSON by default
    }
}
}

```

[Result]

When you run the application and access the endpoint /sample, it returns:

```

{
  "Message": "Hello, World!"
}

```

If you request the same endpoint with Accept: application/xml header, it returns:

```

<AnonymousType>
  <Message>Hello, World!</Message>
</AnonymousType>

```

ASP.NET Core's default JSON support is provided by the System.Text.Json library, which is lightweight and fast. You can also use Newtonsoft.Json if you need more features. Adding XML support is as simple as calling .AddXmlSerializerFormatters() in ConfigureServices. This flexibility allows you to cater to a wide range of client needs without extensive configuration.

[Trivia]

ASP.NET Core's support for multiple data formats is part of its middleware pipeline, which allows you to customize how requests and responses are handled. This modularity is one of the strengths of ASP.NET Core, providing developers with the tools to build highly scalable and maintainable applications.

Serving Static Files with Middleware

ASP.NET Core uses Static Files Middleware to serve static files like images, CSS, and JavaScript, which is essential for delivering web content efficiently.

To serve static files in an ASP.NET Core application, you need to configure the Static Files Middleware. This middleware handles requests for files stored in the wwwroot directory by default.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
namespace StaticFilesExample
{
    public class Startup
    {
        public void Configure(IApplicationBuilder app,
            IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            app.UseStaticFiles(); // Enables serving static files
            app.UseRouting();
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapGet("/", async context =>
                {
                    await context.Response.WriteAsync("Welcome to the Static
Files Example!");
                });
            });
        }
    }
}
```

```
}  
}  
}
```

[Result]

When you place an image file named logo.png in the wwwroot directory and access it via <http://localhost:5000/logo.png>, the image is served directly to the client.

The `UseStaticFiles()` middleware serves files from the wwwroot directory by default. You can customize this behavior by specifying options such as different file providers or request paths. This middleware is crucial for delivering static content efficiently, reducing server load and improving response times.

[Trivia]

Static files are served without any server-side processing, which makes them very efficient. However, it's important to ensure that sensitive files are not exposed through this middleware. ASP.NET Core provides options to restrict access to certain files or directories, enhancing security.

Customizing the Dependency Injection (DI) Container in ASP.NET Core

ASP.NET Core uses a built-in Dependency Injection (DI) container to manage dependencies within your application. However, if you have specific needs, you can replace the default DI container with a custom one.

Below is an example showing how to replace the built-in DI container with a custom container in an ASP.NET Core application.

[Code]

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using MyCustomDIContainer;
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseServiceProviderFactory(new
MyCustomServiceProviderFactory()) // Replace default DI with custom
container
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
// Custom Service Provider Factory
```

```

public class MyCustomServiceProviderFactory :
IServiceProviderFactory<MyCustomContainer>
{
    public MyCustomContainer CreateBuilder(IServiceCollection
services)
    {
        var container = new MyCustomContainer();
        // Configure services here using custom container methods
        return container;
    }
    public IServiceProvider CreateServiceProvider(MyCustomContainer
container)
    {
        return container.BuildServiceProvider();
    }
}

```

[Result]

When you run the application, the custom DI container (MyCustomDIContainer) will handle the service lifetimes and resolve dependencies instead of the default one.

ASP.NET Core's built-in DI container is simple and sufficient for most use cases, but certain scenarios require a more advanced DI framework. This flexibility allows you to integrate complex containers like Autofac, Ninject, or your custom solution. Replacing the DI container involves implementing the `IServiceProviderFactory<TContainerBuilder>` interface and providing your custom implementation, as shown above. The custom container must be able to handle service registrations, resolve services, and manage their lifetimes.

[Trivia]

Some popular DI containers that developers commonly use with ASP.NET Core include Autofac, Ninject, and StructureMap. They offer features like better support for complex scenarios, detailed lifetime management, and more advanced resolution techniques compared to the built-in container.

Handling Multiple Cultures and Languages with Request Localization in ASP.NET Core

ASP.NET Core provides built-in support for request localization, allowing your application to handle multiple cultures and languages, making it easier to create multilingual and culturally aware applications.

Below is an example of how to configure and use request localization to support multiple cultures and languages in an ASP.NET Core application.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.AspNetCore.Localization;
using System.Globalization;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllersWithViews();
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {

```

```

        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    var supportedCultures = new[]
    {
        new CultureInfo("en-US"),
        new CultureInfo("fr-FR"),
        new CultureInfo("es-ES")
    };
    var localizationOptions = new RequestLocalizationOptions
    {
        DefaultRequestCulture = new RequestCulture("en-US"),
        SupportedCultures = supportedCultures,
        SupportedUICultures = supportedCultures
    };
    app.UseRequestLocalization(localizationOptions);
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
}

```

[Result]

When you run the application, it will automatically adjust the culture and language settings based on the request's culture information. For example, if a user accesses the application with a browser set to French (fr-FR), the application will display content in French.

Request localization is essential for creating applications that cater to a global audience. By setting up the `RequestLocalizationOptions`, you can define the default culture and the supported cultures and UI cultures.

ASP.NET Core will then use these settings to manage how content is displayed according to the user's preferences, ensuring that your application is accessible and user-friendly across different languages and regions. It's also important to note that you can customize the culture selection process further by implementing custom logic in the `RequestLocalizationOptions` to handle specific needs.

[Trivia]

ASP.NET Core's localization middleware works seamlessly with data annotations and validation messages, which means that if your application has localized resources for validation errors, these messages will be automatically displayed in the appropriate language based on the request's culture.

Understanding Razor Syntax in ASP.NET Core

Razor syntax is a blend of HTML and C# code that allows developers to create dynamic web pages in ASP.NET Core.

The following example demonstrates how Razor syntax integrates C# logic within an HTML structure to display a personalized greeting message based on user input.

[Code]

```
@{
    var userName = "John";
    var currentTime = DateTime.Now.Hour;
    var greeting = currentTime < 12 ? "Good morning" : "Good
afternoon";
}
<h1>@greeting, @userName!</h1>
<p>The current time is @DateTime.Now.ToString("hh:mm tt").</p>
```

[Result]

Good morning, John!
The current time is 08:30 AM.
(Note: The output will change based on the current system time and the name provided.)

Razor syntax allows for inline C# code within HTML files by using the @ symbol. This makes it easier to manage dynamic content and logic directly in the view. The Razor engine processes this mix of HTML and C# to render the final HTML that is sent to the client's browser. Razor's intelligent parsing helps distinguish between HTML and C# code,

enabling seamless integration of server-side logic into web pages. Understanding this distinction is key to building dynamic and responsive ASP.NET Core applications. In the provided code example: `@{ }` is used to define a code block where you can declare variables and perform logic. `@variableName` injects the value of the variable directly into the HTML. This syntax provides a clear and concise way to blend backend logic with frontend design.

[Trivia]

Razor was first introduced with ASP.NET MVC 3 in 2010 as a simplified and more readable alternative to traditional ASPX view engines. It is now the default view engine in ASP.NET Core, making it a crucial part of modern .NET web development.

36

Preventing CSRF Attacks with Anti-Forgery Tokens in ASP.NET Core

ASP.NET Core employs Anti-Forgery Tokens to protect web applications from Cross-Site Request Forgery (CSRF) attacks.

The following code demonstrates how to implement Anti-Forgery Tokens in an ASP.NET Core form to prevent unauthorized actions by ensuring that each request is legitimate.

[Code]

```
<form asp-controller="Account" asp-action="Login" method="post">
  @Html.AntiForgeryToken()
  <div>
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" />
  </div>
  <div>
    <label for="password">Password:</label>
    <input type="password" id="password" name="password" />
  </div>
  <button type="submit">Login</button>
</form>
```

In the corresponding controller:[HttpPost]

[ValidateAntiForgeryToken]

```
public IActionResult Login(string username, string password)
{
    // Authentication logic here
    return View();
}
```

[Result]

When the form is submitted, ASP.NET Core automatically includes a hidden field containing the Anti-Forgery Token. The `ValidateAntiForgeryToken` attribute in the controller action ensures that the token sent with the request matches the one generated by the server, preventing CSRF attacks. If the tokens do not match, the request is rejected.

Anti-Forgery Tokens are crucial for securing forms in web applications against CSRF attacks, where a malicious user tricks a victim into submitting an unauthorized request. In ASP.NET Core, the `@Html.AntiForgeryToken()` helper generates a hidden form field with a unique token, which is also stored in a cookie. Upon form submission, the server checks that the token in the form matches the one in the cookie. This mechanism ensures that the request is coming from the legitimate user, not a third party. The `ValidateAntiForgeryToken` attribute is used on the server side to enforce this validation. This protection is especially important for forms that perform actions like updating user information, making payments, or changing account settings.

[Trivia]

CSRF attacks are one of the most common vulnerabilities in web applications, as documented by OWASP (Open Web Application Security Project). Implementing Anti-Forgery Tokens is considered a best practice to mitigate these risks, and ASP.NET Core provides this functionality out of the box.

Storing Sessions in ASP.NET Core

In ASP.NET Core, sessions can be stored in memory, a distributed cache, or custom providers, allowing flexibility in how session data is managed.

This example demonstrates how to configure and use session storage in ASP.NET Core using in-memory caching.

[Code]

```
// Startup.cs
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add session services
        services.AddDistributedMemoryCache(); // Use in-memory cache
        for sessions
        services.AddSession(options =>
        {
            options.IdleTimeout = TimeSpan.FromMinutes(30); // Set session
            timeout
            options.Cookie.HttpOnly = true; // Make the session cookie
            HTTP only
            options.Cookie.IsEssential = true; // Make the session cookie
            essential
        });
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            app.UseRouting();
        }
    }
}
```

```

app.UseSession(); // Enable session middleware
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        // Retrieve session value
        int? visitCount = context.Session.GetInt32("VisitCount") ?? 0;
        visitCount++;
        context.Session.SetInt32("VisitCount", visitCount.Value);
        await context.Response.WriteAsync($"You have visited this
page {visitCount} times.");
    });
});
}
}

```

[Result]

You have visited this page 1 times.
 You have visited this page 2 times.
 ...

Sessions in ASP.NET Core allow you to store user data across requests. The `AddDistributedMemoryCache` method configures in-memory caching, which is suitable for development and testing but not recommended for production due to its volatility. The `AddSession` method sets session options like timeout and cookie properties. The `UseSession` middleware must be added before any middleware that reads or writes session data. In this example, the session stores a simple visit count, demonstrating how session data persists across requests.

[Trivia]

In-memory session storage is fast but not suitable for scaled-out applications. For distributed scenarios, consider using Redis or SQL Server as a session store. Custom session providers can be implemented by extending `ISessionStore`.

Understanding ControllerBase in ASP.NET Core

The ControllerBase class in ASP.NET Core serves as the base class for creating Web API controllers, providing essential functionality for handling HTTP requests.

This example illustrates how to create a simple Web API controller using the ControllerBase class in ASP.NET Core.

[Code]

```
// Controllers/SampleController.cs
using Microsoft.AspNetCore.Mvc;
[ApiController]
[Route("[controller]")]
public class SampleController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        return Ok("Hello from ControllerBase!");
    }
}
```

[Result]

Hello from ControllerBase!

The ControllerBase class provides essential methods and properties for handling HTTP requests, such as `Ok()`, `BadRequest()`, and `NotFound()`, which simplify returning standard HTTP responses. It is used when you don't need the full MVC features like views. The `[ApiController]` attribute enhances the controller with features like automatic model

validation and improved routing. The [Route] attribute defines the URL pattern for the controller's actions. In this example, a simple GET endpoint returns a string response.

[Trivia]

While ControllerBase is used for Web APIs, the Controller class is used for MVC applications that require views. The [ApiController] attribute also provides automatic binding source inference, meaning it can automatically determine where to bind parameters from (e.g., query string, route data).

Middleware Execution Order in ASP.NET Core

In ASP.NET Core, middleware components are executed in the exact order they are registered in the Startup class. This order is crucial as it determines how HTTP requests are processed through the pipeline.

The following code example demonstrates how middleware components are registered in the Startup class and how the order affects their execution. We will also include custom middleware to observe this behavior.

[Code]

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            Console.WriteLine("Middleware 1: Before next()");
            await next.Invoke();
            Console.WriteLine("Middleware 1: After next()");
        });
        app.Use(async (context, next) =>
        {
            Console.WriteLine("Middleware 2: Before next()");
            await next.Invoke();
            Console.WriteLine("Middleware 2: After next()");
        });
        app.Run(async context =>
        {
            Console.WriteLine("Middleware 3: Final handler");
            await context.Response.WriteAsync("Hello from Middleware 3!");
        });
    }
}
```

```
    });  
  }  
}
```

[Result]

Middleware 1: Before next()
Middleware 2: Before next()
Middleware 3: Final handler
Middleware 2: After next()
Middleware 1: After next()

In this example, we have three middleware components. The first two are registered using `app.Use`, and the last one is registered with `app.Run`. The `app.Run` method registers a terminal middleware, meaning no other middleware will be executed after this point. Middleware 1 is the first to execute, and it calls the next middleware using `await next.Invoke()`. This continues down the chain to Middleware 2. Middleware 2 then does the same, passing control to Middleware 3, which is the final handler. After Middleware 3 completes, control is returned back up the chain to Middleware 2 and then to Middleware 1, allowing them to perform actions after the `next()` call. The order in which middleware components are registered is critical because it affects the flow of request processing. If, for example, Middleware 2 was registered before Middleware 1, the console output and request processing would be different.

[Trivia]

Middleware is a powerful feature in ASP.NET Core that allows developers to handle requests and responses in a modular way. Middleware components can perform tasks like authentication, logging, error handling, and more. The modular nature of middleware makes it easy to extend or modify the request pipeline without altering the core application logic.

Understanding IActionResult in ASP.NET Core

The IActionResult interface in ASP.NET Core allows controllers to return various types of HTTP responses, enabling flexible and testable action methods.

In the following example, we'll see how a controller can return different IActionResult types, such as Ok(), NotFound(), and BadRequest(), depending on the result of the action.

[Code]

```
public class MyController : ControllerBase
{
    [HttpGet("get-result/{id}")]
    public IActionResult GetResult(int id)
    {
        if (id == 0)
        {
            return BadRequest("Invalid ID provided.");
        }
        else if (id < 0)
        {
            return NotFound("Item not found.");
        }
        else
        {
            return Ok($"Item with ID {id} found.");
        }
    }
}
```

[Result]

For GET /get-result/0, the response would be:
400 Bad Request
Invalid ID provided.
For GET /get-result/-1, the response would be:mathematica
404 Not Found
Item not found.
For GET /get-result/10, the response would be:200 OK
Item with ID 10 found.

The IActionResult interface provides a standardized way for action methods to return various types of HTTP responses. This abstraction is beneficial because it allows developers to easily return different status codes and content types from the same action method. BadRequest() returns a 400 Bad Request status code, indicating that the server cannot or will not process the request due to a client error. NotFound() returns a 404 Not Found status code, indicating that the requested resource could not be found. Ok() returns a 200 OK status code, which indicates that the request has succeeded and the server has returned the requested resource. This approach promotes clean code by avoiding multiple return statements scattered throughout the method and makes unit testing easier by allowing the response type to be controlled and verified.

[Trivia]

IActionResult is just one of several return types supported by ASP.NET Core MVC. Others include JsonResult, ViewResult, and FileResult, each tailored for specific response types like JSON data, HTML views, or file downloads. Using these result types, ASP.NET Core actions can return the appropriate response format based on the request or the action logic.

API Versioning in ASP.NET Core

ASP.NET Core supports API versioning, allowing developers to manage changes to APIs over time without disrupting existing clients.

To implement API versioning in ASP.NET Core, you need to configure your application to recognize different versions of your API. This can be done using the `Microsoft.AspNetCore.Mvc.Versioning` package.

[Code]

```
// Install the package via NuGet Package Manager Console
// PM> Install-Package Microsoft.AspNetCore.Mvc.Versioning
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
        services.AddApiVersioning(options =>
        {
            // Report API versions in response headers
            options.ReportApiVersions = true;
            // Default API version
            options.DefaultApiVersion = new
Microsoft.AspNetCore.Mvc.ApiVersion(1, 0);
            // Assume default version when unspecified
            options.AssumeDefaultVersionWhenUnspecified = true;
        });
    }
    public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
    {

```

```

        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
// Example Controller
using Microsoft.AspNetCore.Mvc;
[ApiController]
[Route("api/v{version:apiVersion}/[controller]")]
public class SampleController : ControllerBase
{
    [HttpGet]
    public IActionResult Get() => Ok("Hello from API version 1.0");
}

```

[Result]

When you access the endpoint `/api/v1.0/sample`, it will return: Hello from API version 1.0.

API versioning is crucial for maintaining backward compatibility while introducing new features. The `Microsoft.AspNetCore.Mvc.Versioning` package provides a flexible way to handle versioning by URL segment, query string, or HTTP header. This approach ensures that clients using older versions of your API can continue to function while you develop new versions.

[Trivia]

API versioning can be combined with Swagger (OpenAPI) to document different API versions, making it easier for developers to understand and use your API.

Logging with ILogger in ASP.NET Core

The ILogger interface in ASP.NET Core is used for logging within the application, providing a standardized way to log messages of various severity levels.

To use the ILogger interface, you need to inject it into your classes and use it to log information, warnings, errors, etc. This example demonstrates basic usage of ILogger in a controller.

[Code]

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    private readonly ILogger<WeatherForecastController> _logger;
    public
WeatherForecastController(ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }
    [HttpGet]
    public IEnumerable<string> Get()
    {
        _logger.LogInformation("WeatherForecastController.Get method
called.");
        return new string[] { "Sunny", "Cloudy", "Rainy" };
    }
}
```

[Result]

When the Get method is called, a log entry is created: Information: WeatherForecastController.Get method called.

The ILogger interface supports logging at different levels: Trace, Debug, Information, Warning, Error, and Critical. This allows developers to filter logs based on the severity level, which is useful for debugging and monitoring applications. ASP.NET Core's logging infrastructure is extensible, allowing you to integrate with various logging providers like Console, Debug, Azure, and more.

[Trivia]

ASP.NET Core's logging system is built on top of Microsoft.Extensions.Logging, which provides a simple and flexible logging abstraction that can be extended with custom logging providers.

Running ASP.NET Core Applications in Docker Containers

Running ASP.NET Core applications inside Docker containers simplifies deployment by encapsulating the application and its dependencies, ensuring consistent environments across different stages of development, testing, and production.

To run an ASP.NET Core application inside a Docker container, you need to create a Dockerfile that specifies how the application should be built and configured inside the container.

[Code]

```
# Use the official ASP.NET Core runtime image
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
# Use the official .NET SDK image to build the app
FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY . .
RUN dotnet restore "MyApp/MyApp.csproj"
RUN dotnet publish "MyApp/MyApp.csproj" -c Release -o /app/publish
# Copy the build output to the runtime image
FROM base AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "MyApp.dll"]
```

[Result]

When building and running the Docker container using the following commands, the application will be hosted inside the container and

```
accessible via the exposed port.docker build -t myapp .
docker run -d -p 8080:80 myapp
Output:arduino
Application is running at http://localhost:8080
```

The Dockerfile is a blueprint that tells Docker how to build and run your application. FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base uses a prebuilt ASP.NET Core runtime image. WORKDIR /app sets the working directory inside the container. EXPOSE 80 tells Docker to expose port 80 to the outside world. The second FROM statement switches to a different image that includes the .NET SDK, which is necessary to build the application. COPY . . copies all files from the host machine to the container. RUN dotnet restore, RUN dotnet publish restore dependencies and compile the app, respectively. Finally, the application is copied to the runtime image and set to start using ENTRYPOINT ["dotnet", "MyApp.dll"]. The key benefit of using Docker with ASP.NET Core is that it allows you to package your application along with all its dependencies in a single, portable container. This makes it easy to deploy the application to any environment that supports Docker without worrying about missing dependencies or mismatched configurations.

[Trivia]

Docker containers are isolated environments that run on a shared operating system, making them lightweight compared to virtual machines. Using Docker with ASP.NET Core ensures consistency between development, testing, and production environments, as the same container image is used across all stages. Docker supports multi-stage builds, which allows you to build your application in one stage and then copy the results into a smaller, more optimized runtime environment.

Understanding ModelState for Input Validation in ASP.NET Core

ModelState in ASP.NET Core is used to validate user input. It provides an easy way to ensure that data submitted by users meets the expected format and rules defined in your application models.

To demonstrate how ModelState works, consider a simple form where a user submits their name and age. We will create a model, apply validation attributes, and then check the ModelState in the controller to handle validation errors.

[Code]

```
// Model: Person.cs
using System.ComponentModel.DataAnnotations;
public class Person
{
    [Required]
    [StringLength(50, MinimumLength = 2)]
    public string Name { get; set; }
    [Range(1, 120)]
    public int Age { get; set; }
}
// Controller: HomeController.cs
using Microsoft.AspNetCore.Mvc;
public class HomeController : Controller
{
    [HttpPost]
    public IActionResult Submit(Person person)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
    }
}
```

```
        return Ok("Submission successful!");  
    }  
}
```

[Result]

When submitting valid data: {

 "Name": "John Doe",

 "Age": 30

}

Response:

Submission successful!

When submitting invalid data: {

 "Name": "",

 "Age": 150

}

Response: {

 "Name": ["The Name field is required."],

 "Age": ["The field Age must be between 1 and 120."]

}

ModelState in ASP.NET Core automatically validates data based on the attributes applied to your model properties. In the provided example: [Required] ensures the Name property is not null or empty. [StringLength(50, MinimumLength = 2)] ensures that the Name is between 2 and 50 characters long. [Range(1, 120)] ensures that the Age is between 1 and 120. When the form is submitted, ASP.NET Core checks the data against these rules. If any rules are violated, ModelState.IsValid will be false, and the validation errors will be returned to the client. This built-in validation mechanism helps ensure that only valid data enters your system, reducing the risk of bugs or unexpected behavior due to invalid inputs.

[Trivia]

ModelState not only checks for validation errors but also tracks if a value was actually submitted, which is helpful for identifying missing fields. You can add custom validation logic by implementing IValidatableObject in your model or by creating custom validation

attributes. ModelState is particularly useful in APIs, where it can return detailed error messages to the client, helping them understand what went wrong with their input.

Handling File Uploads with IFormFile in ASP.NET Core

ASP.NET Core allows you to handle file uploads using the `IFormFile` interface, which provides a straightforward way to process files sent by users through HTTP requests.

The following code demonstrates how to implement file uploads in an ASP.NET Core application using the `IFormFile` interface. This example shows how to accept a file from a form and save it to the server.

[Code]

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using System.IO;
using System.Threading.Tasks;
namespace FileUploadExample.Controllers
{
    public class UploadController : Controller
    {
        [HttpPost("upload")]
        public async Task<ActionResult> UploadFile(IFormFile file)
        {
            if (file == null || file.Length == 0)
                return Content("File not selected");
            var path = Path.Combine(Directory.GetCurrentDirectory(),
"uploads", file.FileName);
            using (var stream = new FileStream(path, FileMode.Create))
            {
                await file.CopyToAsync(stream);
            }
            return Content("File uploaded successfully");
        }
    }
}
```

```
}
```

[Result]

If you upload a file using this controller, you will receive the message: "File uploaded successfully".

The `IFormFile` interface is part of the `Microsoft.AspNetCore.Http` namespace and represents a file sent with the HTTP request. It provides properties like `FileName`, `Length`, and methods like `CopyToAsync` for handling file data efficiently.

In the example, the file is saved to a directory named "uploads" within the current directory. Ensure that this directory exists or handle directory creation in your code. Also, consider security aspects like file size limits and file type validation to prevent malicious uploads.

[Trivia]

ASP.NET Core's model binding system automatically maps form file inputs to `IFormFile` parameters in your controller actions. This feature simplifies handling multipart form data, commonly used in file uploads.

Using UserManager for User Operations in ASP.NET Core Identity

The `UserManager` class in ASP.NET Core Identity provides essential methods for managing user accounts, including creating, updating, and retrieving user information.

The following example showcases how to use the `UserManager` class to create a new user and retrieve user details in an ASP.NET Core Identity setup.

[Code]

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
namespace IdentityExample.Controllers
{
    public class AccountController : Controller
    {
        private readonly UserManager<IdentityUser> _userManager;
        public AccountController(UserManager<IdentityUser>
userManager)
        {
            _userManager = userManager;
        }
        [HttpPost("create")]
        public async Task<ActionResult> CreateUser(string email, string
password)
        {
            var user = new IdentityUser { UserName = email, Email = email
};
            var result = await _userManager.CreateAsync(user, password);
            if (result.Succeeded)
            {
```

```

        return Content("User created successfully");
    }
    return Content("User creation failed: " + string.Join(", ",
result.Errors));
}
[HttpGet("getuser")]
public async Task<IActionResult> GetUser(string email)
{
    var user = await _userManager.FindByEmailAsync(email);
    if (user != null)
    {
        return Content($"User found: {user.UserName}");
    }
    return Content("User not found");
}
}
}

```

[Result]

When creating a user, you will see "User created successfully" if successful. Retrieving a user will display "User found: [username]".

The UserManager class is a core part of ASP.NET Core Identity, designed to handle user-related tasks. It provides asynchronous methods like CreateAsync and FindByEmailAsync, which are crucial for managing user data in a secure and efficient manner.

Ensure that your Identity configuration is set up correctly in the Startup.cs file, including services.AddIdentity and the appropriate database context for storing user information.

[Trivia]

ASP.NET Core Identity is highly extensible, allowing you to customize user properties and roles. The UserManager class can be extended to support additional user operations, making it a versatile tool for authentication and authorization processes.

Introduction to Razor View Engine in ASP.NET Core

The Razor View Engine in ASP.NET Core is a powerful tool that allows developers to generate dynamic HTML content using Razor syntax, which is a combination of HTML and C# code. This engine is commonly used to create views in MVC applications.

Below is a basic example of how to use Razor syntax to generate HTML content within an ASP.NET Core application. The example demonstrates a simple view that displays a list of items dynamically.

[Code]

```
@model List<string>
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Razor View Example</title>
</head>
<body>
  <h1>Item List</h1>
  <ul>
    @foreach (var item in Model)
    {
      <li>@item</li>
    }
  </ul>
</body>
</html>
```

[Result]

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Razor View Example</title>
</head>
<body>
  <h1>Item List</h1>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
  </ul>
</body>
</html>
```

The Razor View Engine seamlessly combines C# code and HTML in the same file. The `@model` directive specifies the type of the model passed to the view, and `@foreach` is used to iterate over the model's data to dynamically generate list items. Razor views are typically part of an MVC application, where controllers pass data to views for rendering. The Razor syntax ensures that C# code is server-side, and the resulting HTML is what is sent to the client.

[Trivia]

Razor is designed to be easy to use and understand, especially for developers familiar with C#. It automatically escapes HTML output to protect against cross-site scripting (XSS) attacks, making it a secure choice for rendering dynamic content.

ASP.NET Core Authentication Middleware Overview

ASP.NET Core's authentication middleware supports a variety of authentication schemes, such as JWT and OAuth, allowing applications to authenticate users through multiple methods depending on the needs of the application.

The following code snippet demonstrates how to configure JWT Bearer authentication in an ASP.NET Core application. This setup is commonly used in APIs that require token-based authentication.

[Code]

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(options =>
    {
        options.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
JwtBearerDefaults.AuthenticationScheme;
    })
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new
TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "yourdomain.com",
            ValidAudience = "yourdomain.com",
```

```

        IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKey"))
    };
});
services.AddControllers();
}
public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
{
    app.UseAuthentication();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
}

```

[Result]

When a client sends a request with a valid JWT token, the server authenticates the user based on the provided token. If the token is valid, the user gains access to protected resources; otherwise, the server responds with an unauthorized error.

JWT (JSON Web Token) is a popular choice for authentication in APIs due to its stateless nature, meaning the server does not need to store session data. The authentication middleware in ASP.NET Core handles token validation, ensuring that requests are authenticated before they reach the endpoint. The `TokenValidationParameters` allow you to define the rules for validating tokens, such as issuer, audience, and signing key. OAuth, on the other hand, is often used for third-party authentication, such as logging in via Google or Facebook.

[Trivia]

ASP.NET Core's authentication system is modular, meaning you can easily switch between different authentication schemes or use multiple schemes simultaneously. For example, you could support both cookie-based authentication and JWT within the same application, depending on the endpoint being accessed.

Authorization Policies in ASP.NET Core

ASP.NET Core allows you to define authorization policies to control access to resources, ensuring that only authorized users can perform certain actions.

Authorization policies in ASP.NET Core are used to enforce security rules for accessing resources. You can define these policies in the Startup.cs file and apply them using attributes in your controllers or actions.

[Code]

```
// Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    // Define an authorization policy
    services.AddAuthorization(options =>
    {
        options.AddPolicy("RequireAdminRole", policy =>
            policy.RequireRole("Admin"));
    });
}

// ExampleController.cs
[Authorize(Policy = "RequireAdminRole")]
public class ExampleController : ControllerBase
{
    [HttpGet]
    public IActionResult GetSecretData()
    {
        return Ok("This is secret data only for Admins!");
    }
}
```

[Result]

When a user with the "Admin" role accesses the GetSecretData endpoint, they will receive the message: "This is secret data only for Admins!" If the user does not have the "Admin" role, they will be denied access.

The AddAuthorization method in Startup.cs is used to define policies. The RequireRole method specifies that the user must have a specific role to satisfy the policy. In the controller, the [Authorize] attribute is used to apply the policy to an action or the entire controller. This ensures that only users with the required role can access the protected resources.

[Trivia]

Authorization policies can be more complex and include multiple requirements, such as claims or custom requirements. They provide a flexible way to manage access control in your application.

Configuring and Starting a Web Application with WebHostBuilder

The WebHostBuilder in ASP.NET Core is a crucial component used to configure and start a web application, setting up essential services and middleware.

WebHostBuilder is responsible for configuring the web host, which includes setting up the server, middleware, and other services required for the application. It is typically used in the Program.cs file to start the application.

[Code]

```
// Program.cs
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

[Result]

When executed, the application starts and listens for incoming HTTP requests. The server is configured based on the settings provided in the Startup class.

WebHostBuilder is a part of the hosting infrastructure in ASP.NET Core. It sets up the Kestrel server by default and configures middleware components defined in the Startup class. The `UseStartup<Startup>()` method specifies the class that contains configuration for services and the request pipeline.

[Trivia]

ASP.NET Core applications can be hosted on various servers, including Kestrel, IIS, and HTTP.sys. The WebHostBuilder allows you to configure which server to use and customize its settings.

Custom Route Constraints in ASP.NET Core

ASP.NET Core allows developers to define custom route constraints, which are rules that determine whether a route should be matched based on specific conditions. This feature enhances URL routing by enabling more control over how requests are processed.

The following code demonstrates how to create a custom route constraint that checks if a route parameter is a valid integer.

[Code]

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using System.Threading.Tasks;
public class IntegerRouteConstraint : IRouteConstraint
{
    public bool Match(HttpContext httpContext,
        Microsoft.AspNetCore.Routing.IRouter route,
        string routeKey,
        RouteValueDictionary values,
        RouteDirection routeDirection)
    {
        {
            return int.TryParse(values[routeKey]?.ToString(), out _);
        }
    }
}
// In Startup.cs
public void Configure(IApplicationBuilder app, IHostingEnvironment
env)
{
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("products/{id:int}", async context =>
```

```
{
    await context.Response.WriteAsync($"Product ID:
{context.Request.RouteValues["id"]}");
});
});
}
```

[Result]

If you access the URL `/products/123`, the response will be:

Product ID: 123

If you access the URL `/products/abc`, there will be a 404 Not Found error.

In this example, we define a custom route constraint called `IntegerRouteConstraint`. The `Match` method checks if the route parameter (in this case, `id`) can be parsed as an integer. If it can, the route is considered a match. The `MapGet` method in the `UseEndpoints` configuration maps the route to a handler that responds with the product ID. If the input is not a valid integer, ASP.NET Core will return a 404 error.

[Trivia]

Custom route constraints can be useful for enforcing specific patterns in your URLs, such as ensuring that IDs are numeric or that certain segments follow a specific format. This can help improve the robustness of your application by preventing invalid requests from being processed.

Using HttpClientFactory in ASP.NET Core

ASP.NET Core's `HttpClientFactory` simplifies the creation and management of `HttpClient` instances, promoting better performance and resource management. It helps avoid common pitfalls such as socket exhaustion.

The following code snippet illustrates how to configure and use `HttpClientFactory` to make an HTTP GET request.

[Code]

```
using Microsoft.Extensions.DependencyInjection;
using System.Net.Http;
using System.Threading.Tasks;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddHttpClient("MyClient", client =>
        {
            client.BaseAddress = new Uri("https://api.example.com/");
            client.DefaultRequestHeaders.Add("Accept", "application/json");
        });
    }
    public async Task<string> GetDataAsync()
    {
        var clientFactory = new HttpClientFactory();
        var client = clientFactory.CreateClient("MyClient");
        var response = await client.GetAsync("data");
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

[Result]

The result of calling `GetDataAsync()` will be the JSON response from the API endpoint, such as:

```
{"key":"value"}
```

(Note: The actual output will depend on the API response.)

In this example, we configure `HttpClientFactory` in the `ConfigureServices` method of the `Startup` class. We create a named client called `MyClient` with a base address and default headers. The `GetDataAsync` method demonstrates how to use this client to make a GET request to an API endpoint. The `EnsureSuccessStatusCode` method throws an exception if the response indicates failure, ensuring that we handle errors appropriately.

[Trivia]

Using `HttpClientFactory` provides several benefits, including automatic management of `HttpClient` lifetimes, built-in support for Polly (a resilience and transient-fault-handling library), and the ability to configure named or typed clients for different purposes. This approach can significantly enhance the maintainability and performance of your applications.

Using GraphQL with ASP.NET Core for Flexible APIs

ASP.NET Core supports GraphQL, allowing developers to create APIs that let clients request only the data they need, improving efficiency and flexibility.

This section demonstrates how to set up a basic GraphQL API in ASP.NET Core. We'll use the GraphQL.NET library to define a simple schema and query.

[Code]

```
// Install the GraphQL.NET package via NuGet
// Command: dotnet add package GraphQL
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using GraphQL;
using GraphQL.Types;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<ISchema, MySchema>();
        services.AddGraphQL(options =>
        {
            options.EnableMetrics = false;
        }).AddSystemTextJson();
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
```

```

        {
            endpoints.MapGraphQL();
        });
    }
}
public class MySchema : Schema
{
    public MySchema(IServiceProvider provider) : base(provider)
    {
        Query = provider.GetRequiredService<MyQuery>();
    }
}
public class MyQuery : ObjectGraphType
{
    public MyQuery()
    {
        Field<StringGraphType>(
            "hello",
            resolve: context => "Hello, GraphQL!"
        );
    }
}
}

```

[Result]

When you run the application and query the GraphQL endpoint with { hello }, the response will be:

```

{
  "data": {
    "hello": "Hello, GraphQL!"
  }
}

```

GraphQL is a query language for APIs and a runtime for executing those queries by using a type system you define for your data. Unlike REST, where the server defines the structure of the response, GraphQL allows the client to specify exactly what data it needs. This can reduce the

amount of data transferred over the network and make applications more efficient.

In the code example, we use GraphQL.NET, a popular library for implementing GraphQL in .NET applications. The MyQuery class defines a simple query that returns a static string. The MySchema class ties everything together, and the Startup class configures the necessary services and middleware.

[Trivia]

GraphQL was developed by Facebook in 2012 and released as an open-source project in 2015. It is widely used by companies like GitHub, Twitter, and Shopify to provide flexible and efficient APIs.

Building Configuration Sources with ConfigurationBuilder in ASP.NET Core

The ConfigurationBuilder class in ASP.NET Core is used to construct configuration sources, allowing applications to retrieve settings from various sources like JSON files, environment variables, and more.

This example illustrates how to use the ConfigurationBuilder to load configuration settings from a JSON file and environment variables.

[Code]

```
using System;
using Microsoft.Extensions.Configuration;
class Program
{
    static void Main(string[] args)
    {
        var builder = new ConfigurationBuilder()
            .SetBasePath(AppContext.BaseDirectory)
            .AddJsonFile("appsettings.json", optional: true, reloadOnChange:
true)
            .AddEnvironmentVariables();
        IConfiguration configuration = builder.Build();
        string settingFromJson = configuration["MySetting"];
        string settingFromEnv = configuration["PATH"];
        Console.WriteLine($"Setting from JSON: {settingFromJson}");
        Console.WriteLine($"Setting from Environment Variable:
{settingFromEnv}");
    }
}
```

[Result]

Assuming appsettings.json contains:

```
{  
  "MySetting": "Value from JSON"  
}
```

The output will be:

Setting from JSON: Value from JSON

Setting from Environment Variable: (value of the PATH environment variable)

The ConfigurationBuilder class in ASP.NET Core is a flexible and powerful way to manage application settings. It allows you to combine multiple configuration sources, such as JSON files, XML files, environment variables, and command-line arguments. This makes it easy to manage different configurations for different environments (e.g., development, staging, production).

In this example, we load settings from a JSON file named appsettings.json and environment variables. The SetBasePath method sets the base path for the configuration files, and AddJsonFile specifies the JSON file to load. The AddEnvironmentVariables method includes environment variables in the configuration.

[Trivia]

ASP.NET Core's configuration system is designed to work seamlessly with dependency injection. You can inject IConfiguration into your services to access configuration settings directly, making it easy to manage application settings across different layers of your application.

Using Environment Variables in ASP.NET Core

Environment variables in ASP.NET Core are used to manage settings that vary between different environments, such as development, testing, and production.

Environment variables allow you to configure your application without changing the code. They are particularly useful for managing sensitive information like API keys and connection strings.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using System;
var builder = WebApplication.CreateBuilder(args);
// Access environment variables
var mySetting =
Environment.GetEnvironmentVariable("MY_SETTING") ??
"DefaultSetting";
builder.Services.AddControllers();
var app = builder.Build();
app.MapGet("/", () => $"My Setting: {mySetting}");
app.Run();
To set an environment variable, you can use the command line:
export MY_SETTING=ProductionSetting
```

[Result]

When you run the application and access the root URL, it will display:
My Setting: ProductionSetting



Environment variables are key-value pairs that are stored outside your application. In ASP.NET Core, they can be accessed using `Environment.GetEnvironmentVariable()`. This allows you to keep sensitive information secure and change configurations without modifying the codebase. ASP.NET Core also supports configuration providers that can read from various sources, including environment variables, JSON files, and more.

[Trivia]

In ASP.NET Core, the `IConfiguration` interface is used to access configuration settings. It can automatically read from environment variables, which are prefixed with the application name by default. This makes it easy to manage configurations across different deployment environments.

JWT Bearer Tokens for API Authentication in ASP.NET Core

ASP.NET Core supports JSON Web Tokens (JWT) for stateless authentication in APIs, allowing secure data transmission between parties.

JWT is a compact, URL-safe means of representing claims to be transferred between two parties. It is widely used for API authentication as it enables stateless, scalable security.

[Code]

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;
var builder = WebApplication.CreateBuilder(args);
// Configure JWT authentication
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new
TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "your-issuer",
            ValidAudience = "your-audience",
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("your-secret-key"))
        };
    });
```

```
builder.Services.AddControllers();  
var app = builder.Build();  
app.UseAuthentication();  
app.UseAuthorization();  
app.MapControllers();  
app.Run();
```

To issue a JWT token, you would typically have an endpoint that generates a token upon successful login.

[Result]

The application will require a valid JWT token for accessing protected endpoints. If a valid token is not provided, the response will be:
HTTP 401 Unauthorized

JWTs consist of three parts: Header, Payload, and Signature. The Header typically consists of the token type and the signing algorithm. The Payload contains the claims, which are statements about an entity (typically, the user) and additional data. The Signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way. In ASP.NET Core, the `JwtBearer` middleware handles the validation of these tokens.

[Trivia]

JWT tokens are self-contained, meaning they carry all the information needed for authentication, which makes them ideal for stateless authentication in distributed systems. However, because they are self-contained, they can grow large and should be used judiciously to avoid performance issues.

Understanding Dependency Injection Scopes in ASP.NET Core

Dependency Injection (DI) in ASP.NET Core is a design pattern used to achieve Inversion of Control (IoC) between classes and their dependencies. ASP.NET Core provides three DI scopes: Singleton, Scoped, and Transient. Understanding these scopes is crucial for correctly managing the lifecycle and behavior of your services.

In this section, we will explore each of the three DI scopes—Singleton, Scoped, and Transient—by providing examples and explaining when to use each scope.

[Code]

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using System;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Singleton: A single instance throughout the application's lifetime
        services.AddSingleton<ISingletonService, SingletonService>();
        // Scoped: A single instance per request
        services.AddScoped<IScopedService, ScopedService>();
        // Transient: A new instance every time it is requested
        services.AddTransient<ITransientService, TransientService>();
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
```



```

    {
        app.UseDeveloperExceptionPage();
    }
    app.Run(async context =>
    {
        var singletonService =
context.RequestServices.GetService<ISingletonService>();
        var scopedService =
context.RequestServices.GetService<IScopedService>();
        var transientService =
context.RequestServices.GetService<ITransientService>();
        await context.Response.WriteAsync($"Singleton:
{singletonService.GetHashCode()}\n");
        await context.Response.WriteAsync($"Scoped:
{scopedService.GetHashCode()}\n");
        await context.Response.WriteAsync($"Transient:
{transientService.GetHashCode()}\n");
    });
}
}
public interface ISingletonService { }
public interface IScopedService { }
public interface ITransientService { }
public class SingletonService : ISingletonService { }
public class ScopedService : IScopedService { }
public class TransientService : ITransientService { }

```

[Result]

When running the above code and making multiple requests to the server, the output might look like this: Singleton: 12345678

Scoped: 87654321

Transient: 11223344

If you refresh the page multiple times, you will see that the Singleton service retains the same instance (hash code) across all requests, the Scoped service retains the same instance within a single request but changes across different requests, and the Transient service has a new instance for every retrieval.

Singleton: The Singleton scope ensures that only one instance of the service is created and shared across all requests and all users. This is ideal for stateless services or those that maintain some shared state or resource.Scoped: The Scoped scope ensures that a single instance of the service is created and shared within the same request. Each new request gets a new instance of the service. This is particularly useful for database contexts in web applications, ensuring that changes are isolated to a single request.Transient: The Transient scope ensures that every time the service is requested, a new instance is created. This is suitable for lightweight, stateless services that do not maintain any shared state.Understanding these scopes is vital for controlling the behavior of your application and ensuring that resources are used efficiently and correctly. Misusing these scopes can lead to issues like memory leaks, race conditions, or unexpected behavior.

[Trivia]

In ASP.NET Core, services registered with Singleton scope should be thread-safe because they are shared across multiple requests, which may occur simultaneously. Scoped and Transient services, on the other hand, can avoid thread-safety concerns as they are isolated to a single request or operation.

Mastering Model Validation in ASP.NET Core

Model validation in ASP.NET Core ensures that data submitted by users meets the required criteria before it is processed. This is primarily achieved through data annotations and custom validation attributes, which enforce rules and provide error messages when the data does not conform to expectations.

We will demonstrate how to use built-in data annotations for model validation and how to create custom validation attributes in ASP.NET Core.

[Code]

```
using Microsoft.AspNetCore.Mvc;
using System.ComponentModel.DataAnnotations;
public class UserModel
{
    [Required(ErrorMessage = "Name is required.")]
    [StringLength(50, ErrorMessage = "Name cannot be longer than 50
characters.")]
    public string Name { get; set; }
    [Range(18, 99, ErrorMessage = "Age must be between 18 and 99.")]
    public int Age { get; set; }
    [EmailAddress(ErrorMessage = "Invalid email address.")]
    public string Email { get; set; }
}
public class CustomValidationAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
ValidationContext validationContext)
    {
        var user = (UserModel)validationContext.ObjectInstance;
        if (user.Age < 21 && user.Email.EndsWith("example.com"))
```

```

        {
            return new ValidationResult("Users under 21 cannot use an
'example.com' email address.");
        }
        return ValidationResult.Success;
    }
}
[ApiController]
[Route("api/[controller]")]
public class UsersController : ControllerBase
{
    [HttpPost]
    public IActionResult CreateUser([FromBody] UserModel user)
    {
        if (ModelState.IsValid)
        {
            return Ok("User is valid");
        }
        return BadRequest(ModelState);
    }
}

```

[Result]

If the user submits data that does not meet the validation criteria, the API will return a 400 Bad Request with validation error messages. For example: {

```

    "Name": ["Name is required."],
    "Age": ["Age must be between 18 and 99."],
    "Email": ["Invalid email address."]
}

```

If the custom validation fails, a message like the following would be included: {

```

    "Email": ["Users under 21 cannot use an 'example.com' email
address."]
}

```

Data Annotations: These are attributes that can be applied to model properties to enforce validation rules. Common annotations include [Required], [StringLength], [Range], and [EmailAddress]. These annotations automatically integrate with ASP.NET Core's model binding and validation mechanisms. **Custom Validation:** When the built-in annotations do not meet your needs, you can create custom validation attributes by inheriting from `ValidationAttribute` and overriding the `IsValid` method. This allows for complex validation logic that can cross multiple fields or involve external services. Model validation is crucial in web applications as it prevents invalid data from being processed, which could lead to errors, security vulnerabilities, or inconsistent data. Always validate user input both on the client-side and server-side to ensure robust application behavior.

[Trivia]

The `IValidatableObject` interface can be implemented in a model to perform validation logic that requires access to multiple fields at once. This is an alternative to custom validation attributes and is useful when validation logic is dependent on the combination of several fields within a model.

Accessing Application Settings with IConfiguration in ASP.NET Core

The IConfiguration interface in ASP.NET Core is used to access application settings, allowing developers to retrieve configuration values from various sources like JSON files, environment variables, and more.

This example demonstrates how to use the IConfiguration interface to access a setting from an appsettings.json file in an ASP.NET Core application.

[Code]

```
// appsettings. {
  "AppSettings": {
    "SiteTitle": "My Awesome Site"
  }
}
// Startup.cs
using Microsoft.Extensions.Configuration;
public class Startup
{
    public IConfiguration Configuration { get; }
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }
    public void ConfigureServices(IServiceCollection services)
    {
        // Access the configuration setting
        string siteTitle = Configuration["AppSettings:SiteTitle"];
        Console.WriteLine($"Site Title: {siteTitle}");
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
```

```
{  
    // Other configurations...  
}  
}
```

[Result]

Site Title: My Awesome Site

The IConfiguration interface provides a simple way to access configuration settings in a hierarchical manner. In the example, the appsettings.json file contains a section called AppSettings with a key SiteTitle. The Configuration property in Startup class is used to retrieve this value using the key path AppSettings:SiteTitle. This approach allows for easy management of configuration settings across different environments by simply changing the source file or environment variables.

[Trivia]

ASP.NET Core supports multiple configuration providers, which means you can load configuration from various sources such as JSON, XML, INI files, environment variables, command-line arguments, and even custom providers. The configuration system is flexible and allows for overriding values from one source with another, following a specific order of precedence.

Handling Exceptions with Custom Exception Filters in ASP.NET Core

ASP.NET Core allows you to create custom exception filters to handle specific exceptions in a centralized manner, providing a way to manage error handling and logging across your application.

This example illustrates how to create and use a custom exception filter to handle exceptions in an ASP.NET Core application.

[Code]

```
// CustomExceptionHandler.cs
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using System;
public class CustomExceptionHandler : IExceptionHandler
{
    public void OnException(ExceptionContext context)
    {
        if (context.Exception is InvalidOperationException)
        {
            context.Result = new ContentResult
            {
                Content = "An invalid operation occurred.",
                StatusCode = 400
            };
        }
    }
}
// Startup.cs
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
```



```

        services.AddControllers(options =>
        {
            options.Filters.Add<CustomExceptionFilter>();
        });
    }
    public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
    {
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
// ExampleController.cs
using Microsoft.AspNetCore.Mvc;
[ApiController]
[Route("[controller]")]
public class ExampleController : ControllerBase
{
    [HttpGet]
    public IActionResult Get()
    {
        throw new InvalidOperationException("This is a test exception.");
    }
}

```

[Result]

An invalid operation occurred.

Custom exception filters in ASP.NET Core provide a way to handle exceptions at a global level. In the example, the `CustomExceptionFilter` class implements the `IExceptionHandler` interface and checks if the exception is of type `InvalidOperationException`. If it is, it sets a custom response with a status code of 400. This filter is added to the MVC pipeline in `Startup.cs`, ensuring that any controller action throwing an `InvalidOperationException` will result in a consistent error response. This

approach helps in maintaining clean and readable code by separating error handling logic from business logic.

[Trivia]

Exception filters are part of the ASP.NET Core middleware pipeline, and they are executed after the action method is called but before the result is returned. They are ideal for handling exceptions that occur during the execution of controller actions. However, they won't catch exceptions that occur outside of the MVC action pipeline, such as those thrown by middleware.

Understanding View Components in ASP.NET Core

View Components in ASP.NET Core are used to encapsulate reusable chunks of UI, allowing developers to create modular and maintainable web applications.

View Components are similar to partial views but provide more functionality. They allow you to execute server-side logic and return a view. This makes them ideal for rendering dynamic content like navigation menus, tag clouds, or sidebars.

[Code]

```
// Step 1: Create a View Component class
public class PriorityListViewComponent : ViewComponent
{
    public IViewComponentResult Invoke(int maxPriority, bool isDone)
    {
        // Sample data to simulate a database or business logic
        var items = new List<string> { "Task 1", "Task 2", "Task 3" }
            .Where(x => x.Length <= maxPriority && x.Contains(isDone ?
"Done" : "Task")).ToList();
        return View(items);
    }
}

// Step 2: Create a View for the View Component
// File: Views/Shared/Components/PriorityList/Default.cshtml
@model List<string>
<ul>
    @foreach (var item in Model)
    {
        <li>@item</li>
    }
</ul>
```

```
// Step 3: Invoke the View Component in a Razor View
// File: Views/Home/Index.cshtml
@await Component.InvokeAsync("PriorityList", new { maxPriority =
10, isDone = false })
```

[Result]

The result will be an unordered list of tasks that match the criteria specified in the Invoke method.

View Components are powerful because they allow you to encapsulate both logic and presentation in a single reusable unit. Unlike partial views, View Components can have parameters and can execute logic before rendering the view. They are ideal for scenarios where you need to render dynamic content that depends on some server-side processing.

[Trivia]

View Components do not use model binding, which means you have to pass parameters explicitly. They are not intended to replace controllers but to complement them by providing a way to render parts of a page that require server-side logic.

Using ObjectResult in ASP.NET Core

ASP.NET Core's `ObjectResult` is used to return a serialized object with a specific status code, enabling developers to provide detailed HTTP responses.

`ObjectResult` is a type of `ActionResult` in ASP.NET Core that allows you to return an object serialized to the response format negotiated by the client. It is useful for returning data from API endpoints with a specific HTTP status code.

[Code]

```
// Step 1: Create a Controller with an Action Method
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        // Simulate fetching a product from a database
        var product = new { Id = id, Name = "Sample Product", Price = 99.99 };
        if (product == null)
        {
            return NotFound(new { Message = "Product not found" });
        }
        return new ObjectResult(product)
        {
            StatusCode = 200 // OK
        };
    }
}
```

[Result]

When a valid product ID is provided, the result will be a JSON object representing the product with a 200 OK status code. If the product is not found, a 404 Not Found status with an error message is returned.

ObjectResult is versatile because it allows you to specify the HTTP status code and the object to be serialized in the response. This makes it a good choice for API responses where you need to provide both data and status information. It uses content negotiation to determine the best format to return the object, such as JSON or XML.

[Trivia]

ObjectResult is part of the ASP.NET Core MVC framework and is commonly used in Web API controllers. It can be customized further by setting headers or using different formatters to control how the response is serialized.

Understanding the IFormCollection Interface in ASP.NET Core

The IFormCollection interface in ASP.NET Core is used to handle form data submitted by users in a web application. It represents a collection of form data key-value pairs where the key is the name of the form field and the value is the submitted data.

In this example, we demonstrate how to use the IFormCollection interface to access form data within an ASP.NET Core controller action.

[Code]

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
namespace MyApp.Controllers
{
    public class FormController : Controller
    {
        [HttpPost]
        public IActionResult Submit(IFormCollection formData)
        {
            // Access form data using the key of the form field
            string name = formData["name"];
            string email = formData["email"];
            // Return a simple message with the submitted data
            return Content($"Name: {name}, Email: {email}");
        }
    }
}
```

[Result]

If the form submission includes a field with the name "name" and "email", the response will display: Name: John Doe, Email: john.doe@example.com

The `IFormCollection` interface is essentially a specialized dictionary that allows you to access form data by key. When a form is submitted via a POST request, the data is sent to the server, and the `IFormCollection` interface provides a way to read this data in your controller. It's particularly useful when you need to handle forms where the number of fields or their names are not known at compile time. Each entry in the `IFormCollection` is a key-value pair where the key is the form field's name and the value is the user input. Moreover, `IFormCollection` also supports file uploads, as it includes a property called `Files`, which is an `IFormFileCollection` containing the files uploaded through the form. This makes `IFormCollection` a versatile tool for handling various types of form submissions in ASP.NET Core.

[Trivia]

`IFormCollection` is case-insensitive when it comes to form keys, meaning that "name" and "Name" would be treated as the same key. ASP.NET Core's model binding automatically handles simple types, but `IFormCollection` is particularly useful when working with dynamic forms or when you need to manually process the form data.

CORS Support in ASP.NET Core

ASP.NET Core provides built-in support for Cross-Origin Resource Sharing (CORS), a security feature that allows or restricts resources to be accessed from different domains. This is crucial for modern web applications where APIs might be consumed by client-side applications hosted on different domains.

In this example, we'll see how to enable CORS in an ASP.NET Core application to allow requests from specific origins.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
namespace MyApp
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            // Add CORS services and configure a policy
            services.AddCors(options =>
            {
                options.AddPolicy("AllowSpecificOrigin",
                    builder => builder.WithOrigins("https://example.com")
                        .AllowAnyMethod()
                        .AllowAnyHeader());
            });
        }
        public void Configure(IApplicationBuilder app,
            IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
```

```

    {
        app.UseDeveloperExceptionPage();
    }
    app.UseRouting();
    // Apply the CORS policy globally
    app.UseCors("AllowSpecificOrigin");
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
}
}

```

[Result]

When a request is made from "https://example.com" to your ASP.NET Core application, the CORS policy will allow it, provided that the request method and headers are allowed by the policy.

CORS is vital for security in web applications because it prevents unauthorized domains from accessing resources. However, in certain situations, like when building an API meant to be consumed by different clients across multiple domains, you may need to explicitly enable CORS. In ASP.NET Core, you can configure CORS at a global level (for the entire application) or at the endpoint level. The configuration involves defining a CORS policy that specifies which origins are allowed, which HTTP methods are permitted, and which headers can be used in requests. You can also customize CORS further by adding more restrictive or lenient policies, depending on the security needs of your application. The flexibility of ASP.NET Core's CORS implementation allows developers to fine-tune how cross-origin requests are handled.

[Trivia]

The CORS protocol is a part of the web standard and is enforced by browsers, meaning that it only affects requests made by browsers. Non-browser clients like Postman or cURL can bypass CORS. CORS can be a complex topic, particularly when dealing with preflight requests (OPTIONS) which are automatically sent by browsers for certain types

of cross-origin requests. Understanding how preflight requests work is crucial for debugging CORS issues.

Using TempData in ASP.NET Core

TempData in ASP.NET Core is used to store data temporarily, making it available for the next request.

The following code demonstrates how to use TempData in an ASP.NET Core controller to pass data between actions.

[Code]

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        // Storing data in TempData
        TempData["Message"] = "Hello, this is a TempData message!";
        return RedirectToAction("Display");
    }
    public IActionResult Display()
    {
        // Retrieving data from TempData
        var message = TempData["Message"] as string;
        return Content(message); // Displays: Hello, this is a TempData
message!
    }
}
```

[Result]

Hello, this is a TempData message!

TempData is a dictionary-like storage mechanism in ASP.NET Core that allows you to store data that persists for a single request. It is particularly useful when you need to pass data between actions during a redirect. TempData uses session state to store data, which means it requires

session middleware to be enabled. Unlike ViewData and ViewBag, which are used for passing data to views, TempData is specifically for passing data between requests. It's important to note that TempData is cleared after it is read, making it ideal for scenarios where data should only be available once.

[Trivia]

TempData internally uses the ITempDataProvider interface, which by default, relies on cookies or session state to persist data. You can customize this behavior by implementing your own ITempDataProvider.

Generating URLs with IUrlHelper in ASP.NET Core

IUrlHelper in ASP.NET Core is used to generate URLs for routes, actions, and files, facilitating navigation and linking.

The following code demonstrates how to use the IUrlHelper interface in an ASP.NET Core controller to generate URLs for different actions.

[Code]

```
public class HomeController : Controller
{
    private readonly IUrlHelper _urlHelper;
    public HomeController(IUrlHelper urlHelper)
    {
        _urlHelper = urlHelper;
    }
    public IActionResult Index()
    {
        // Generating a URL for the 'Display' action
        string url = _urlHelper.Action("Display", "Home");
        return Content($"Generated URL: {url}");
    }
    public IActionResult Display()
    {
        return Content("This is the Display action.");
    }
}
```

[Result]

Generated URL: /Home/Display

The `IUrlHelper` interface in ASP.NET Core provides methods to generate URLs based on routing information. This is crucial for creating links in your application that are robust to changes in routing configuration. The `Action` method generates a URL for a specified action method, taking the action name and optionally the controller name and route values. Using `IUrlHelper` helps avoid hardcoding URLs, making your application more maintainable and adaptable to changes. It is automatically available in controllers through dependency injection.

[Trivia]

`IUrlHelper` is part of the ASP.NET Core MVC framework and is typically accessed via the `Url` property in a controller or view. It also provides methods like `RouteUrl` for generating URLs based on route names and `Content` for resolving virtual paths to application-relative paths.

Configuring Session State in ASP.NET Core

Learn how to configure session state in ASP.NET Core, including setting timeouts and choosing storage mechanisms.

This example demonstrates how to configure session state in an ASP.NET Core application. You'll see how to set up session timeouts and select a storage mechanism for session data.

[Code]

```
// Startup.cs
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add session services to the service collection
        services.AddDistributedMemoryCache(); // Use in-memory cache
        for session storage
        services.AddSession(options =>
        {
            options.IdleTimeout = TimeSpan.FromMinutes(30); // Set session
            timeout to 30 minutes
            options.Cookie.HttpOnly = true; // Make the session cookie
            HTTP only
            options.Cookie.IsEssential = true; // Make the session cookie
            essential
        });
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {

```



```

        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();
    app.UseSession(); // Enable session middleware
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
}

```

[Result]

The application will store session data in memory and sessions will timeout after 30 minutes of inactivity.

In ASP.NET Core, session state is a feature that allows you to store user data while they browse your application. By default, session state is not enabled, so you need to configure it in the Startup.cs file. You can choose different storage mechanisms for session data, such as in-memory, distributed cache, or SQL Server. The IdleTimeout property determines how long a session can remain inactive before it is abandoned. The session cookie settings ensure that cookies are secure and essential for the application's functionality.

[Trivia]

Session state is useful for storing user-specific data that needs to persist across multiple requests, such as shopping cart contents or user

preferences. However, it is important to manage session data carefully to avoid excessive memory usage and ensure data security.

Adding Real-Time Functionality with ASP.NET SignalR

Discover how to use ASP.NET SignalR to add real-time communication features to your ASP.NET Core applications.

This example illustrates how to integrate ASP.NET SignalR into an ASP.NET Core application to enable real-time functionality, such as live chat or notifications.

[Code]

```
// Startup.cs
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
        services.AddSignalR(); // Add SignalR services
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();
```

```

        app.UseAuthorization();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
            endpoints.MapHub<ChatHub>("/chatHub"); // Map the SignalR
hub
        });
    }
}
// ChatHub.cs
using Microsoft.AspNetCore.SignalR;
public class ChatHub : Hub
{
    public async Task SendMessage(string user, string message)
    {
        await Clients.All.SendAsync("ReceiveMessage", user, message); //
Broadcast message to all clients
    }
}

```

[Result]

Clients connected to the /chatHub endpoint will receive real-time messages sent by other clients.

ASP.NET SignalR is a library that simplifies adding real-time web functionality to applications. It enables server-side code to push content to connected clients instantly, rather than having the server wait for a client to request new data. SignalR supports WebSockets, and falls back to other techniques for older browsers. The ChatHub class in this example is a hub that manages connections and messaging. The SendMessage method broadcasts messages to all connected clients using the Clients.All.SendAsync method.

[Trivia]

Real-time functionality is crucial for applications like chat apps, live notifications, and online gaming. SignalR abstracts the complexities of real-time communication, allowing developers to focus on building

interactive features without worrying about the underlying transport mechanisms.

Response Compression in ASP.NET Core

ASP.NET Core supports response compression to reduce the size of HTTP responses, which improves the performance of web applications by decreasing the amount of data transferred between the server and client.

The following example demonstrates how to configure response compression in an ASP.NET Core application.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.ResponseCompression;
using System.IO.Compression;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddResponseCompression(options =>
        {
            options.EnableForHttps = true; // Enable compression for HTTPS
requests
            options.Providers.Add<GzipCompressionProvider>(); // Add
Gzip compression provider
        });
        services.Configure<GzipCompressionProviderOptions>(options =>
        {
            options.Level = CompressionLevel.Fastest; // Set the
compression level to Fastest
        });
    }
}
```

```

public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseResponseCompression(); // Use the response compression
middleware
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("Hello, World!"); //
Sample response
        });
    });
}
}

```

[Result]

When this application is run and accessed via a browser, the response "Hello, World!" will be compressed using Gzip before being sent to the client. The size of the response will be smaller than if compression were not enabled.

Response compression in ASP.NET Core can significantly enhance the performance of web applications by reducing the amount of data that needs to be transmitted over the network. This is particularly beneficial for clients with slower internet connections or when large amounts of data need to be sent. The `AddResponseCompression` method adds the response compression services to the service collection. By default, ASP.NET Core supports Gzip and Brotli compression. The `GzipCompressionProviderOptions` allows you to specify the compression level, where `CompressionLevel.Fastest` provides the quickest compression time with reasonable compression ratios. It's important to

note that response compression is particularly useful for text-based responses like HTML, CSS, JavaScript, and JSON. However, binary formats like images or already compressed files like PDFs and ZIPs won't benefit much from additional compression.

[Trivia]

In production environments, it's recommended to test the impact of response compression on the performance and CPU usage, as higher compression levels can increase server load. ASP.NET Core also allows selective compression based on MIME types, so you can optimize which content types to compress.

Database Configuration by Environment in ASP.NET Core

ASP.NET Core applications can be configured to use different databases based on the environment, such as development, staging, or production, allowing developers to seamlessly switch between configurations during the development cycle.

The following example demonstrates how to configure different databases for different environments in an ASP.NET Core application using Entity Framework Core.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
public class Startup
{
    private readonly IConfiguration _configuration;
    public Startup(IConfiguration configuration)
    {
        _configuration = configuration;
    }
    public void ConfigureServices(IServiceCollection services)
    {
        // Configuring different databases based on the environment
        services.AddDbContext<MyDbContext>(options =>
        {
            var connectionString =
            _configuration.GetConnectionString("DefaultConnection");
            if (_configuration.GetValue<string>
("ASPNETCORE_ENVIRONMENT") == "Development")
```

```

        {
            options.UseSqlite(connectionString); // Use SQLite in
development
        }
        else if (_configuration.GetValue<string>
("ASPNETCORE_ENVIRONMENT") == "Staging")
        {
            options.UseSqlServer(connectionString); // Use SQL Server in
staging
        }
        else if (_configuration.GetValue<string>
("ASPNETCORE_ENVIRONMENT") == "Production")
        {
            options.UseNpgsql(connectionString); // Use PostgreSQL in
production
        }
    });
}
public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers(); // Assuming controllers are used
    });
}
}
public class MyDbContext : DbContext
{
    public MyDbContext(DbContextOptions<MyDbContext> options)
        : base(options)
    {
    }
    // Define DbSet for your entities
    public DbSet<MyEntity> MyEntities { get; set; }
}

```

```
}  
public class MyEntity  
{  
    public int Id { get; set; }  
    public string Name { get; set; }  
}
```

[Result]

Depending on the environment in which the application is running (Development, Staging, or Production), the application will connect to different databases. For instance, in development, it might use an SQLite database, while in production, it connects to a PostgreSQL database.

ASP.NET Core's configuration system is highly flexible, allowing you to adjust settings based on the environment the application is running in. The environment can be specified via the `ASPNETCORE_ENVIRONMENT` environment variable. Common values are Development, Staging, and Production, but you can define custom environments as needed. The `IConfiguration` interface provides access to configuration settings from various sources, including JSON files, environment variables, and command-line arguments. This allows for seamless switching between different configurations without modifying the codebase, which is particularly useful for CI/CD pipelines and managing separate development, staging, and production environments. The `UseSqlite`, `UseSqlServer`, and `UseNpgsql` methods configure the Entity Framework Core context to use different database providers based on the environment. This approach enables developers to work with lightweight, easy-to-reset databases like SQLite during development while deploying to more robust databases like SQL Server or PostgreSQL in production.

[Trivia]

It's a good practice to store sensitive configuration settings, such as connection strings, in environment-specific configuration files like `appsettings.Development.json`, `appsettings.Staging.json`, and `appsettings.Production.json`. Additionally, sensitive information can be stored in environment variables or in Azure Key Vault, making the application more secure and easier to manage in different environments.

Understanding Endpoint Routing in ASP.NET Core

Endpoint routing in ASP.NET Core allows you to delay routing decisions until later in the pipeline, offering greater flexibility and control in your application's request processing.

In ASP.NET Core, endpoint routing is a mechanism that maps incoming requests to the appropriate handlers, but unlike traditional routing, it allows these decisions to be made later in the processing pipeline. This feature is beneficial for scenarios where routing needs to be influenced by middleware.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRouting();
    }
    public void Configure(IApplicationBuilder app)
    {
        var routeBuilder = new RouteBuilder(app);
        // Define routes
        routeBuilder.MapGet("hello", context =>
        {
            return context.Response.WriteAsync("Hello World!");
        });
        var routes = routeBuilder.Build();
    }
}
```

```

        // Use routing middleware
        app.UseRouter(routes);
        app.Run(async (context) =>
        {
            await context.Response.WriteAsync("No matching route
found.");
        });
    }
}
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}

```

[Result]

When you navigate to <http://localhost:5000/hello>, the result will be: Hello World!

If you navigate to any other URL, the result will be: No matching route found.

In this example, the routing middleware is set up using the `MapGet` method, which maps the URL pattern "hello" to a handler that returns "Hello World!" to the client. This demonstrates the core idea of endpoint routing: the routing decision is delayed until the `UseRouter` middleware processes the request, allowing more flexible handling of requests. Traditionally, routing decisions were made early in the pipeline, which limited flexibility, especially when working with middleware that needed to influence or even alter routing behavior. With endpoint

routing, the middleware can now participate in or alter routing decisions, making it a powerful tool for complex applications. This feature is particularly useful in scenarios where you need fine-grained control over request processing, such as in API versioning, localization, or implementing custom logic that depends on certain request characteristics.

[Trivia]

Endpoint routing was introduced in ASP.NET Core 2.2 as a step towards more modular and flexible routing capabilities. It integrates deeply with the middleware pipeline, and from ASP.NET Core 3.0 onwards, it has become the standard approach, replacing traditional routing in most scenarios. Understanding the pipeline and how routing fits into it is crucial for building scalable and maintainable applications.

Managing Forms with ASP.NET Core's FormTagHelper

ASP.NET Core's FormTagHelper simplifies the management of form elements in Razor views, ensuring proper generation of HTML attributes and facilitating model binding.

The FormTagHelper in ASP.NET Core helps in automatically generating the correct attributes for forms, such as the action and method attributes, while also supporting the seamless integration of model data into form elements.

[Code]

```
@model MyApp.Models.UserModel
<form asp-action="Register" method="post">
    <label asp-for="Username"></label>
    <input asp-for="Username" class="form-control" />
    <label asp-for="Email"></label>
    <input asp-for="Email" class="form-control" />
    <button type="submit">Register</button>
</form>
csharp
using Microsoft.AspNetCore.Mvc;
namespace MyApp.Controllers
{
    public class AccountController : Controller
    {
        [HttpPost]
        public IActionResult Register(UserModel model)
        {
            if (ModelState.IsValid)
            {
                // Save data or perform other actions
                return RedirectToAction("Success");
            }
        }
    }
}
```

```

        }
        return View(model);
    }
}
}
csharp
namespace MyApp.Models
{
    public class UserModel
    {
        public string Username { get; set; }
        public string Email { get; set; }
    }
}

```

[Result]

When rendered, the Razor view generates HTML like:

```

<form
action="/Account/Register" method="post">
    <label for="Username">Username</label>
    <input type="text" id="Username" name="Username" class="form-
control" value="" />
    <label for="Email">Email</label>
    <input type="email" id="Email" name="Email" class="form-control"
value="" />
    <button type="submit">Register</button>
</form>

```

The `FormTagHelper` automatically binds the form's `asp-action` attribute to the controller action specified, ensuring the form posts to the correct endpoint. It also generates the correct name and id attributes for the input elements, facilitating easy model binding and validation. `FormTagHelper` also works with other helpers like `InputTagHelper` and `LabelTagHelper` to generate proper HTML for form controls. This tight integration simplifies the process of creating forms, particularly when dealing with model binding and validation, as it ensures the form elements are correctly wired up to your backend model properties. This feature significantly reduces the potential for errors that can arise from manually

coding these attributes and helps maintain consistency across forms in your application.

[Trivia]

FormTagHelper and other TagHelpers were introduced in ASP.NET Core to enhance the productivity and consistency of Razor view development. By allowing server-side code to influence the structure and attributes of HTML elements, TagHelpers bridge the gap between C# code and HTML markup, making Razor views more powerful and easier to maintain.

File Streaming in ASP.NET Core Using FileStreamResult

File streaming in ASP.NET Core allows you to efficiently send files to clients without loading the entire file into memory. The `FileStreamResult` class is used to return a file from a stream, which is particularly useful for large files.

The following code demonstrates how to use `FileStreamResult` to stream a file from the server to the client. This example assumes you have a file stored on the server that you want to send to the user.

[Code]

```
using Microsoft.AspNetCore.Mvc;
using System.IO;
public class FileController : Controller
{
    public IActionResult DownloadFile()
    {
        var filePath = "path/to/your/file.txt"; // Specify the path to your file
        var fileStream = new FileStream(filePath, FileMode.Open,
FileAccess.Read);
        var fileName = "downloadedFile.txt"; // The name the user will see
when downloading
        return File(fileStream, "application/octet-stream", fileName);
    }
}
```

[Result]

When the `DownloadFile` action is invoked, the browser will prompt the user to download `downloadedFile.txt` containing the contents of `file.txt`.

In this example, we create a `FileStream` to read the specified file. The `File` method of the `Controller` class is then used to return a `FileStreamResult`. The parameters include the stream, the content type (in this case, `application/octet-stream` for binary files), and the filename for the download prompt. This method is efficient because it streams the file directly from disk to the response without loading it entirely into memory.

[Trivia]

Memory Efficiency: Streaming files is crucial for performance, especially with large files, as it minimizes memory usage.

Content Types: Adjust the content type based on the file type (e.g., `application/pdf` for PDF files).

Error Handling: Always implement error handling to manage cases where the file may not exist or cannot be accessed.

Configuring Cookies with ASP.NET Core's CookieOptions

ASP.NET Core provides the `CookieOptions` class to configure various properties of cookies, such as expiration, path, and security settings. This is essential for managing user sessions and preferences.

The following code snippet illustrates how to set cookie options when creating a cookie in an ASP.NET Core application.

[Code]

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
public class CookieController : Controller
{
    public IActionResult SetCookie()
    {
        var cookieOptions = new CookieOptions
        {
            Path = "/",
            HttpOnly = true, // Prevents JavaScript access to the cookie
            Secure = true, // Ensures the cookie is sent over HTTPS
            Expires = DateTimeOffset.UtcNow.AddDays(7) // Cookie expires
in 7 days
        };
        Response.Cookies.Append("MyCookie", "CookieValue",
cookieOptions);
        return Content("Cookie has been set!");
    }
}
```

[Result]

When the SetCookie action is invoked, a cookie named MyCookie with the value CookieValue is created and sent to the client's browser.

In this example, we create an instance of CookieOptions to define how the cookie should behave. The Path property specifies the URL path for which the cookie is valid. The HttpOnly property helps mitigate the risk of client-side script accessing the cookie, enhancing security. The Secure property ensures that the cookie is only sent over HTTPS connections, while the Expires property sets the lifespan of the cookie. This approach is essential for maintaining user sessions securely.

[Trivia]

Cookie Security: Always use HttpOnly and Secure flags for cookies that contain sensitive information.

SameSite Attribute: Consider setting the SameSite attribute to prevent CSRF attacks by controlling how cookies are sent with cross-site requests.

Cookie Size Limit: Be aware that cookies have a size limit (typically around 4KB), so avoid storing large amounts of data.

Configuring Custom Razor View Locations in ASP.NET Core

In ASP.NET Core, you can customize the locations where Razor views are stored, allowing you to structure your project in a way that suits your needs.

You can configure custom Razor view locations in ASP.NET Core by modifying the `RazorViewEngineOptions` in your `Startup.cs` file. This flexibility allows you to organize your views outside the default `Views` folder.

[Code]

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    // Configure custom Razor view locations
    services.Configure<RazorViewEngineOptions>(options =>
    {
        options.ViewLocationFormats.Clear();
        options.ViewLocationFormats.Add("/CustomViews/{1}/{0}.cshtml"); // Adds a custom location
        options.ViewLocationFormats.Add("/MoreViews/{1}/{0}.cshtml"); // Adds another custom location
        options.ViewLocationFormats.Add("/Views/Shared/{0}.cshtml"); // Keeps the shared view location
    });
}
```

[Result]

If the above configuration is applied, ASP.NET Core will first look for views in the custom locations

(/CustomViews/ControllerName/ViewName.cshtml and /MoreViews/ControllerName/ViewName.cshtml) before falling back to the default locations.

The `RazorViewEngineOptions.ViewLocationFormats` is a list where each entry specifies a path format to search for views. The placeholders `{0}` and `{1}` represent the view name and controller name, respectively. Clearing the `ViewLocationFormats` and adding custom paths allows complete control over where views are located. This feature is useful in large projects or when adhering to specific project structures.

[Trivia]

In ASP.NET Core, Razor views are usually searched for in the Views directory, but you can also include additional or entirely different folders using this configuration. This customization can help when migrating from older MVC frameworks where the view folder structure might differ.

Understanding the ActionResult Class in ASP.NET Core

The ActionResult class in ASP.NET Core represents the outcome of an action method, determining what the framework will return to the client.

The ActionResult class is a base class for all action results in ASP.NET Core. It provides various derived classes like ViewResult, JsonResult, RedirectResult, etc., each representing different types of responses.

[Code]

```
public IActionResult MyAction()
{
    if (someCondition)
    {
        return View(); // Returns a ViewResult
    }
    else if (anotherCondition)
    {
        return RedirectToAction("AnotherAction"); // Returns a
RedirectToActionResult
    }
    else
    {
        return NotFound(); // Returns a NotFoundResult
    }
}
```

[Result]

Depending on the conditions, the action method may return different results: If someCondition is true, a view will be returned. If

anotherCondition is true, the user will be redirected to another action. If neither condition is met, a 404 Not Found result will be returned.

The IActionResult interface allows a single action method to return multiple types of results, providing flexibility in how the response is handled. For example, a method might return a ViewResult to render a page, a JsonResult to return JSON data, or a RedirectResult to redirect the user to another URL. This polymorphism allows the developer to tailor responses dynamically based on the logic within the method.

[Trivia]

The ActionResult<T> introduced in ASP.NET Core 2.1 allows you to return either an ActionResult or a specific type, like ActionResult<MyModel>. This is particularly useful in APIs where you want to return a model or a status code, such as Ok(myModel) or NotFound().

Dynamic Compilation of Razor Views in ASP.NET Core

In ASP.NET Core, dynamic compilation of Razor views allows developers to modify view files without needing to restart the application. This feature is particularly useful during development, as it speeds up the feedback loop when making changes to the UI.

The following code demonstrates how to enable dynamic compilation for Razor views in an ASP.NET Core application.

[Code]

```
// In the Startup.cs file, add the following in the ConfigureServices
method
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews()
        .AddRazorRuntimeCompilation(); // Enable runtime compilation
}
// In the Views folder, create a simple Razor view (e.g., Index.cshtml)
@{
    ViewData["Title"] = "Dynamic Compilation Example";
}
<h1>@ViewData["Title"]</h1>
<p>This view supports dynamic compilation!</p>
```

[Result]

When you modify the Index.cshtml file and refresh the browser, the changes will be reflected immediately without restarting the application.

Dynamic compilation is enabled by adding the `AddRazorRuntimeCompilation()` method in the `ConfigureServices`

method of the Startup.cs file. This allows the application to compile Razor views at runtime, which is particularly beneficial during the development phase.

The Razor view engine processes .cshtml files, and with runtime compilation enabled, any changes made to these files will be detected and compiled on-the-fly. This eliminates the need for a full application restart, allowing for rapid iteration and testing of UI changes.

[Trivia]

Dynamic compilation is not recommended for production environments due to performance overhead. It is primarily intended for development scenarios.

To use this feature, ensure that you have the `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation` NuGet package installed in your project.

Implementing Custom Logic with IActionFilter in ASP.NET Core

The IActionFilter interface in ASP.NET Core allows developers to execute custom logic before and after action methods are invoked. This is useful for cross-cutting concerns like logging, authentication, or modifying the action's result.

The following example illustrates how to create a custom action filter using the IActionFilter interface.

[Code]

```
// Create a custom action filter by implementing IActionFilter
public class CustomActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Logic before the action executes
        Console.WriteLine("Before action executes: " +
context.ActionDescriptor.DisplayName);
    }
    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Logic after the action executes
        Console.WriteLine("After action executes: " +
context.ActionDescriptor.DisplayName);
    }
}
// Register the filter globally in Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews(options =>
    {
        options.Filters.Add<CustomActionFilter>(); // Add the custom filter
```

```
});  
}
```

[Result]

When an action method is invoked, the console will display messages indicating the execution order of the action filter:

Before action executes:

YourNamespace.Controllers>YourController>YourAction

After action executes:

YourNamespace.Controllers>YourController>YourAction

The `IActionFilter` interface provides two methods: `OnActionExecuting` and `OnActionExecuted`. The former is called just before the action method runs, while the latter is called just after the action method completes.

In the example above, we create a custom action filter named `CustomActionFilter` that logs messages to the console before and after the execution of any action method. This filter is then registered globally, meaning it will apply to all controllers and actions in the application. Using action filters can help maintain clean code by separating concerns, allowing you to handle repetitive tasks in a centralized manner.

[Trivia]

You can also create filters that implement `IAsyncActionFilter` for asynchronous action methods.

Filters can be applied at the controller level or action level, providing flexibility in how they are used across your application.

Understanding IFormFileCollection in ASP.NET Core

The IFormFileCollection interface in ASP.NET Core represents a collection of files uploaded through a form. It allows developers to handle multiple file uploads easily.

The following code demonstrates how to use the IFormFileCollection interface to process uploaded files in an ASP.NET Core application.

[Code]

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
[ApiController]
[Route("api/[controller]")]
public class FileUploadController : ControllerBase
{
    [HttpPost("upload")]
    public IActionResult UploadFiles(IFormFileCollection files)
    {
        if (files.Count == 0)
        {
            return BadRequest("No files were uploaded.");
        }
        foreach (var file in files)
        {
            // Here you can process each file (e.g., save to disk)
            // For demonstration, we will just return the file name
            Console.WriteLine($"Uploaded file: {file.FileName}");
        }
        return Ok("Files uploaded successfully.");
    }
}
```

[Result]

If you send a POST request to `/api/fileupload/upload` with files attached, the output will be:

Uploaded file: example1.txt

Uploaded file: example2.jpg

Files uploaded successfully.

The `IFormFileCollection` interface allows you to manage multiple files uploaded in a single request. Each file can be accessed through the collection, enabling you to perform operations like saving to a server or processing the file contents. The `files.Count` property gives you the number of files uploaded, and you can iterate through each file using a `foreach` loop.

[Trivia]

`IFormFileCollection` is part of the `Microsoft.AspNetCore.Http` namespace.

It is commonly used in scenarios where users need to upload multiple files, such as image galleries or document submissions.

Each `IFormFile` object in the collection provides properties like `FileName`, `Length`, and `ContentType` to help you manage the uploaded files effectively.

Standardizing API Responses with API Conventions in ASP.NET Core

API conventions in ASP.NET Core allow developers to define standard responses for their APIs, making it easier to maintain consistency across different endpoints.

The following code snippet illustrates how to implement API conventions to standardize responses in an ASP.NET Core application.

[Code]

```
using Microsoft.AspNetCore.Mvc;
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpGet("{id}")]
    [ProducesResponseType(typeof(Product), StatusCodes.Status200OK)]
    [ProducesResponseType(StatusCodes.Status404NotFound)]
    public IActionResult GetProduct(int id)
    {
        var product = GetProductById(id); // Assume this method fetches a
product
        if (product == null)
        {
            return NotFound();
        }
        return Ok(product);
    }
}
```

[Result]

If you send a GET request to `/api/products/1` and the product exists, the output will be:

```
{
  "id": 1,
  "name": "Sample Product",
  "price": 19.99
}
```

If the product does not exist, the output will be:
404 Not Found

In the code example, we define API conventions using the `ProducesResponseType` attribute. This attribute specifies the type of response that the action method can produce, along with the corresponding HTTP status codes. By doing this, we ensure that consumers of the API know what to expect when they make requests, improving the overall developer experience.

[Trivia]

API conventions help document the API automatically, making it easier for developers to understand how to interact with it.

You can define global API conventions by configuring them in the `Startup.cs` file, allowing you to apply them across all controllers.

Using `ProducesResponseType` is particularly useful for generating Swagger documentation, which can be invaluable for API consumers.

Custom Authorization with AuthorizationHandler in ASP.NET Core

The `AuthorizationHandler` class in ASP.NET Core is used to create custom authorization policies that allow developers to enforce specific rules and conditions when accessing resources. This approach provides fine-grained control over authorization logic, which is useful for complex applications requiring detailed access controls.

In this example, we will create a custom authorization policy using `AuthorizationHandler` in ASP.NET Core. This policy checks if a user meets a specific requirement, such as belonging to a particular role or having a certain claim.

[Code]

```
// Requirement class
public class CustomRequirement : IAuthorizationRequirement
{
    public string RequiredRole { get; }
    public CustomRequirement(string requiredRole)
    {
        RequiredRole = requiredRole;
    }
}

// AuthorizationHandler class
public class CustomAuthorizationHandler :
    AuthorizationHandler<CustomRequirement>
{
    protected override Task
    HandleRequirementAsync(AuthorizationHandlerContext context,
        CustomRequirement requirement)
    {
        if (context.User.IsInRole(requirement.RequiredRole))
        {
```

```

        context.Succeed(requirement);
    }
    return Task.CompletedTask;
}
}
// Registering the AuthorizationHandler in Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy("CustomPolicy", policy =>
            policy.Requirements.Add(new CustomRequirement("Admin")));
    });
    services.AddSingleton<IAuthorizationHandler,
CustomAuthorizationHandler>();
}

```

[Result]

The result of this code is that when a user attempts to access a resource protected by the "CustomPolicy" policy, the CustomAuthorizationHandler will check if the user is in the "Admin" role. If the user meets this condition, access will be granted.

AuthorizationHandler is a crucial part of ASP.NET Core's flexible authorization system. It allows for the implementation of complex authorization logic that can consider multiple factors, such as user roles, claims, or other contextual data. By using this class, developers can centralize and reuse authorization logic across different parts of an application, reducing redundancy and improving maintainability. The IAuthorizationRequirement interface represents the requirements that must be met for the authorization policy to succeed. Each AuthorizationHandler instance is responsible for determining if the given context fulfills these requirements. The custom authorization policies you define using AuthorizationHandler can be applied at the controller or action level using the [Authorize(Policy = "CustomPolicy")] attribute. This approach is particularly useful when you need to enforce different rules depending on the user's role, claim, or other properties.

[Trivia]

Authorization in ASP.NET Core can also be combined with resource-based authorization, where the authorization logic depends on the specific resource being accessed. This is achieved by adding an additional `AuthorizationHandler` that takes both the requirement and the resource as input.

Global Exception Handling with Middleware in ASP.NET Core

ASP.NET Core supports middleware to handle exceptions globally, allowing developers to manage errors in a centralized location. This reduces the need for repetitive error handling code throughout the application and ensures consistent error responses.

In this example, we will configure a global exception handler using middleware in an ASP.NET Core application. This middleware will catch all unhandled exceptions, log them, and return a standardized error response.

[Code]

```
// Custom middleware for global exception handling
public class ExceptionHandlingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<ExceptionHandlingMiddleware> _logger;
    public ExceptionHandlingMiddleware(RequestDelegate next,
    ILogger<ExceptionHandlingMiddleware> logger)
    {
        _next = next;
        _logger = logger;
    }
    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "An unhandled exception occurred.");
        }
    }
}
```

```

        context.Response.StatusCode =
(int)HttpStatusCode.InternalServerError;
        context.Response.ContentType = "application/json";
        await context.Response.WriteAsync(new
        {
            StatusCode = context.Response.StatusCode,
            Message = "Internal Server Error"
        }.ToString());
    }
}
}
// Registering the middleware in Startup.cs
public void Configure(IApplicationBuilder app)
{
    app.UseMiddleware<ExceptionHandlerMiddleware>();
    // Other middleware registrations
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}

```

[Result]

When an unhandled exception occurs during the processing of a request, the `ExceptionHandlerMiddleware` will catch it, log the error, and return a 500 Internal Server Error response with a JSON message. This ensures that all exceptions are handled in a consistent manner across the application.

Middleware in ASP.NET Core is a powerful feature that allows you to manage the request and response pipeline effectively. By placing exception handling in middleware, you ensure that all unhandled exceptions are caught and processed in a centralized location. This approach not only simplifies error management but also enhances the security and reliability of your application by preventing exceptions from leaking sensitive information to the client. The middleware pattern in ASP.NET Core is highly modular, meaning you can stack multiple

middleware components in a sequence. The order of these components is critical, as it determines how requests and responses flow through the pipeline. Centralized exception handling also makes it easier to log errors consistently, implement retry logic, or customize error responses based on the client type (e.g., returning detailed error messages in development environments and generic ones in production).

[Trivia]

In addition to custom middleware, ASP.NET Core provides built-in exception handling middleware such as `UseDeveloperExceptionPage` for development environments and `UseExceptionHandler` for production. These can be used alongside or instead of custom middleware, depending on the specific needs of your application.

Razor Pages Simplify Page-Focused Development in ASP.NET Core

Razor Pages are a feature of ASP.NET Core that make it easier to build web applications by focusing on individual pages rather than controllers and actions. This approach streamlines the development process, especially for page-centric scenarios.

The following code example demonstrates how to create a simple Razor Page that displays a greeting message. This example will help beginners understand the structure and functionality of Razor Pages.

[Code]

```
// Pages/Greet.cshtml.cs
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
public class GreetModel : PageModel
{
    public string Message { get; private set; }
    public void OnGet()
    {
        Message = "Hello, welcome to ASP.NET Core Razor Pages!";
    }
}
<!-- Pages/Greet.cshtml -->
@page
@model GreetModel
<!DOCTYPE html>
<html>
<head>
    <title>Greeting Page</title>
</head>
<body>
    <h1>@Model.Message</h1>
```



```
</body>  
</html>
```

[Result]

When you navigate to /Greet, the output will be:
Hello, welcome to ASP.NET Core Razor Pages!

In this example, we define a Razor Page by creating two files: Greet.cshtml.cs and Greet.cshtml. The Greet.cshtml.cs file contains the page model, which handles the page's logic. The OnGet method is called when the page is accessed via a GET request, and it sets the Message property. The Greet.cshtml file is the view that renders the HTML. The @model directive links the view to the page model, allowing us to access the Message property directly.

[Trivia]

Razor Pages were introduced in ASP.NET Core 2.0 to provide a more straightforward way to develop web applications. They encourage a page-based programming model, making it easier for developers to manage and maintain their code. Razor Pages also support features like model binding, validation, and dependency injection, which enhance their functionality.

Using StatusCodes Class for HTTP Status Codes in ASP.NET Core

In ASP.NET Core, the StatusCodes class provides a set of constants that represent common HTTP status codes. This makes it easier to return appropriate responses from your web applications.

The following example demonstrates how to use the StatusCodes class to return different HTTP status codes based on the outcome of an operation, such as a successful request or an error.

[Code]

```
// Controllers/HomeController.cs
using Microsoft.AspNetCore.Mvc;
public class HomeController : Controller
{
    public IActionResult Index()
    {
        // Simulating a successful operation
        return Ok("Operation successful!");
    }
    public IActionResult Error()
    {
        // Simulating an error
        return StatusCode(StatusCode.Status500InternalServerError, "An
error occurred.");
    }
}
```

[Result]

When you navigate to /Home/Index, the output will be:
Operation successful!

When you navigate to /Home/Error, the output will be:
An error occurred.

In this example, we have a HomeController with two actions: Index and Error. The Index action returns an HTTP 200 OK status with a success message using the `Ok()` method. The Error action simulates an error by returning a 500 Internal Server Error status using the `StatusCode()` method along with the `StatusCodes.Status500InternalServerError` constant. This approach makes it clear what HTTP status is being returned, improving code readability and maintainability.

[Trivia]

The `StatusCodes` class in ASP.NET Core is part of the `Microsoft.AspNetCore.Http` namespace and includes constants for all standard HTTP status codes. Using these constants helps avoid magic numbers in your code, making it clearer and easier to understand. Additionally, ASP.NET Core automatically handles many status codes, simplifying error handling and response generation in web applications.

Using IApplicationLifetime for Application Startup and Shutdown in ASP.NET Core

The `IApplicationLifetime` interface in ASP.NET Core allows you to execute custom code during the startup and shutdown of your application. This can be useful for tasks such as initializing resources when the application starts or gracefully shutting down connections when the application stops.

In this example, we will use `IApplicationLifetime` to run a simple piece of code during the application startup and shutdown.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using System;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add services to the container.
    }
    public void Configure(IApplicationBuilder app,
        IHostApplicationLifetime appLifetime)
    {
        // Code to run on application startup
        appLifetime.ApplicationStarted.Register(() =>
        {
            Console.WriteLine("Application has started.");
        });
    }
}
```

```

// Code to run on application shutdown
appLifetime.ApplicationStopping.Register(() =>
{
    Console.WriteLine("Application is stopping...");
});
appLifetime.ApplicationStopped.Register(() =>
{
    Console.WriteLine("Application has stopped.");
});
// Configure the HTTP request pipeline.
app.UseRouting();
app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/", async context =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
});
}
}

```

[Result]

When you run this application, the following output will be displayed in the console:

```

mathematica
Application has started.
Application is stopping...
Application has stopped.

```

The `IApplicationLifetime` interface provides three key events:

- `ApplicationStarted`: Triggered after the application starts, useful for any logic that should execute right after the app is up and running.
- `ApplicationStopping`: Triggered when the application is shutting down but hasn't fully stopped. This is a good place to dispose of resources or notify other services that the application is shutting down.
- `ApplicationStopped`: Triggered when the application has completely shut down.

The `IApplicationLifetime` interface is particularly useful in services and background tasks where you might need to start or

clean up resources (e.g., closing database connections, stopping background tasks) during application lifecycle events.

[Trivia]

The `IAApplicationLifetime` interface was replaced by `IHostApplicationLifetime` in ASP.NET Core 3.0. The newer interface offers the same functionality but is part of the `Microsoft.Extensions.Hosting` namespace, which is aligned with the generic host model in .NET Core.

Enabling Request Body Buffering in ASP.NET Core

ASP.NET Core supports request buffering, allowing you to read the request body multiple times. This is particularly useful in scenarios such as logging request data or validating the request body multiple times without losing the original data.

In this example, we will enable request buffering and demonstrate how the request body can be read multiple times within a single request.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using System.IO;
using System.Text;
using System.Threading.Tasks;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add services to the container.
    }
    public void Configure(IApplicationBuilder app,
        IHostApplicationLifetime appLifetime)
    {
        app.Use(async (context, next) =>
        {
            context.Request.EnableBuffering(); // Enable request buffering
            // Read the request body as a string
            var reader = new StreamReader(context.Request.Body,
                Encoding.UTF8);
```

```

        string requestBody = await reader.ReadToEndAsync();
        Console.WriteLine("Request Body First Read: " + requestBody);
        // Reset the request body stream position
        context.Request.Body.Position = 0;
        // Read the request body again
        requestBody = await reader.ReadToEndAsync();
        Console.WriteLine("Request Body Second Read: " +
requestBody);
        // Reset the request body stream position before next middleware
        context.Request.Body.Position = 0;
        await next.Invoke();
    });
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapPost("/", async context =>
        {
            await context.Response.WriteAsync("Request Processed.");
        });
    });
}
}

```

[Result]

When you send a POST request with a body to this application, the console output will show: Request Body First Read: {your request body content}

Request Body Second Read: {your request body content}

By default, ASP.NET Core does not allow the request body to be read multiple times because it would require buffering the data in memory or on disk. However, by calling `context.Request.EnableBuffering()`, you enable buffering for the request body, allowing it to be read multiple times. This is particularly important when the request body needs to be read in different middlewares or when you need to log the request body while still allowing it to be processed by the controller or further down

the pipeline. The `Stream.Position` property is used to reset the stream to the beginning so that it can be read again.

[Trivia]

Buffering large request bodies in memory can cause performance issues and out-of-memory exceptions. ASP.NET Core allows you to configure the size limit for buffering and, in cases where the body is too large, it can write the buffer to a file instead of keeping it in memory. This behavior can be controlled through middleware options and configuration settings.

Understanding ActionResult<T> in ASP.NET Core

ActionResult<T> is a return type in ASP.NET Core that allows action methods to return a specific type or a different result type, such as HTTP status codes.

Below is an example of an action method using ActionResult<T>, which returns a Product object if found, or a NotFound result if not.

[Code]

```
using Microsoft.AspNetCore.Mvc;
public class ProductsController : ControllerBase
{
    private readonly List<Product> _products = new List<Product>
    {
        new Product { Id = 1, Name = "Laptop", Price = 999.99 },
        new Product { Id = 2, Name = "Smartphone", Price = 699.99 }
    };
    [HttpGet("{id}")]
    public ActionResult<Product> GetProduct(int id)
    {
        var product = _products.FirstOrDefault(p => p.Id == id);
        if (product == null)
        {
            return NotFound();
        }
        return product;
    }
}
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
```

```
public double Price { get; set; }  
}
```

[Result]

If a product with the specified id is found, the JSON representation of the product is returned. If no product is found, a 404 Not Found status is returned.

`ActionResult<T>` is a powerful feature in ASP.NET Core because it allows you to combine the return of an object with an HTTP status code. This pattern simplifies the controller logic, as it avoids the need to manually construct responses for different scenarios. For example, in the code provided, if the product is found, it is returned directly as a JSON object. If not, the `NotFound()` method is used to return a 404 HTTP status code. This makes the code cleaner and easier to maintain. Using `ActionResult<T>` also makes it easier to unit test your controller methods, as you can verify both the type of the result and the actual content returned.

[Trivia]

The `ActionResult<T>` type is part of the ASP.NET Core 2.1 release and later. Before `ActionResult<T>` was introduced, developers would typically return `ActionResult` and have to cast objects into specific types manually, making the code more cumbersome.

Using HttpContext in ASP.NET Core

HttpContext in ASP.NET Core provides access to information about the current HTTP request, including request headers, query strings, and user details.

The following code demonstrates how to access various properties of HttpContext in an ASP.NET Core controller.

[Code]

```
using Microsoft.AspNetCore.Mvc;
public class InfoController : ControllerBase
{
    [HttpGet("request-info")]
    public IActionResult GetRequestInfo()
    {
        var request = HttpContext.Request;
        var response = new
        {
            Method = request.Method,
            Path = request.Path,
            QueryString = request.QueryString.ToString(),
            UserAgent = request.Headers["User-Agent"].ToString()
        };
        return Ok(response);
    }
}
```

[Result]

A JSON object containing the HTTP method, request path, query string, and User-Agent header is returned when the request-info endpoint is accessed.

HttpContext is a key class in ASP.NET Core that represents all HTTP-specific information about an individual HTTP request. It's available in every controller and middleware, providing access to both the request and response objects, session state, user information, and more. The HttpContext.Request property allows you to inspect details about the incoming request, such as the HTTP method (GET, POST, etc.), request path, headers, and query strings. Meanwhile, HttpContext.Response lets you modify the outgoing response, such as setting headers or status codes. HttpContext is also central to middleware development, where you might need to intercept or modify the request or response as it passes through the pipeline. Understanding HttpContext is crucial for tasks such as authentication, authorization, logging, and handling custom request processing.

[Trivia]

HttpContext is not thread-safe, meaning you should avoid storing it in long-lived objects or static fields. Always use it within the scope of the current request to prevent unexpected behavior. In ASP.NET Core, HttpContext is injected into controllers and middleware by the framework, ensuring it is scoped correctly to the current request.

Configuring Form Data Processing with ASP.NET Core's FormOptions Class

The FormOptions class in ASP.NET Core is used to configure the behavior of form data processing, such as setting limits on the size of form data and the number of allowed form elements.

To demonstrate how to configure form data processing in ASP.NET Core using the FormOptions class, we will create a simple example that limits the maximum allowed size of form data.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http.Features;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.Configure<FormOptions>(options =>
        {
            // Set the maximum allowed size of form data to 10 MB
            options.MultipartBodyLengthLimit = 10 * 1024 * 1024; // 10 MB
            options.BufferBody = false;
        });
    }
    public void Configure(IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
    }
}
```

```

    }
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("FormOptions
configuration example.");
        });
    });
}
}

```

[Result]

When the application is run, it starts a web server that applies the FormOptions configuration. No direct output is visible from this configuration, but form data submissions exceeding 10 MB will be rejected by the server.

The FormOptions class provides several properties that control how ASP.NET Core handles form data. The MultipartBodyLengthLimit property sets the maximum size for multipart form data, which is commonly used for file uploads. Setting this limit helps prevent denial-of-service (DoS) attacks by restricting the amount of data the server processes. The BufferBody property determines whether the server buffers the entire request body. Setting it to false is useful when dealing with large uploads, as it avoids storing the entire body in memory. It's important to note that these settings are global for the application. If you need more granular control, consider setting these options at the middleware level or implementing custom middleware.

[Trivia]

In addition to MultipartBodyLengthLimit, the FormOptions class has other useful properties like MemoryBufferThreshold, which defines the size limit before the form data is written to a disk-based buffer, and ValueCountLimit, which limits the number of key-value pairs that can be processed in a form.

Passing Data to Views Using ViewData in ASP.NET Core

The ViewData dictionary in ASP.NET Core allows you to pass data from controllers to views. It's a flexible way to transfer information that doesn't need to be strongly typed.

To illustrate how to use ViewData in ASP.NET Core, we'll create a simple controller action that passes a message to a view.

[Code]

```
using Microsoft.AspNetCore.Mvc;
public class HomeController : Controller
{
    public IActionResult Index()
    {
        // Passing data to the view using ViewData
        ViewData["Message"] = "Hello, ViewData!";
        return View();
    }
}
```

And in the corresponding Razor view file (Index.cshtml):

```
@{
    // Retrieving data from ViewData
    var message = ViewData["Message"] as string;
}
<h1>@message</h1>
```

[Result]

When you run the application and navigate to the corresponding route, the browser will display:
Hello, ViewData!

ViewData is a dictionary provided by ASP.NET Core for passing data between controllers and views. It allows you to pass data as key-value pairs, where the keys are strings and the values are objects. While ViewData provides flexibility, it lacks the strong typing of ViewBag or view models, which can make it less safe and more error-prone. ViewData is ideal for passing small amounts of data or for scenarios where the data structure is not known until runtime. However, if you need to pass complex data structures or want to benefit from IntelliSense and compile-time checking, consider using a view model instead.

[Trivia]

ViewData and ViewBag are similar, but ViewData is a dictionary, while ViewBag is a dynamic object that provides a more convenient syntax in some cases. Both are stored in the same underlying dictionary, so changes made to ViewData are reflected in ViewBag and vice versa.

Graceful Shutdown in ASP.NET Core

ASP.NET Core provides a feature known as graceful shutdown, which allows the application to finish processing ongoing requests before it stops. This is essential for maintaining a good user experience and ensuring data integrity.

The following code demonstrates how to implement graceful shutdown in an ASP.NET Core application by configuring the host to allow for a shutdown timeout.

[Code]

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            })
            .ConfigureServices(services =>
            {
                // Configure services here
            });
}
```

To enable graceful shutdown, you can configure the `KestrelServerOptions` in the `Startup.cs` file:

```
public void ConfigureServices(IServiceCollection services)
{

```

```
services.Configure<KestrelServerOptions>(options =>
{
    options.ShutdownTimeout = TimeSpan.FromSeconds(30); // Set
shutdown timeout
});
}
```

[Result]

When the application is stopped (for example, via Ctrl+C in the terminal), it will wait for up to 30 seconds to finish processing any ongoing requests before shutting down.

Graceful shutdown is crucial for web applications, especially those that handle transactions or long-running processes. By configuring a shutdown timeout, you ensure that users do not experience abrupt disconnections or loss of data. This feature is particularly useful in production environments where uptime and reliability are paramount.

[Trivia]

ASP.NET Core uses the `IHostApplicationLifetime` interface to manage the application's lifecycle events, including shutdown.

You can also hook into the `ApplicationStopping` event to perform any cleanup tasks before the application stops.

Generating URLs with `Url.Action` in ASP.NET Core

The `Url.Action` method in ASP.NET Core is used to generate a fully qualified URL for a specific action method in a controller. This is particularly useful for creating links in views or for redirects.

The following code example shows how to use the `Url.Action` method to create a URL to a specific action in a controller.

[Code]

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        // Generate a URL for the About action in the HomeController
        string url = Url.Action("About", "Home");
        // Return the generated URL as a view result
        return Content($"The URL to the About page is: {url}");
    }
    public IActionResult About()
    {
        return Content("This is the About page.");
    }
}
```

[Result]

When you navigate to the `Index` action of the `HomeController`, it will display a message with the URL to the `About` page, such as:
The URL to the `About` page is: `http://localhost:5000/Home/About`

The `Url.Action` method constructs a URL based on the routing configuration of your ASP.NET Core application. It takes two parameters: the name of the action method and the name of the controller. If you need to include route values or query parameters, you can pass an anonymous object as a third parameter. This method is particularly useful for creating links that are resilient to changes in the routing configuration.

[Trivia]

The generated URL can include additional route values, such as parameters, making it flexible for various scenarios.

You can use `Url.Action` in Razor views, controllers, and even in JavaScript by passing the URL to a script.

Limiting Request Size in ASP.NET Core

ASP.NET Core allows developers to set limits on the size of HTTP requests to protect their applications from excessively large payloads, which could otherwise lead to denial of service (DoS) attacks or resource exhaustion.

Below is an example of how to configure the maximum allowed request body size in an ASP.NET Core application using middleware.

[Code]

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<FormOptions>(options =>
    {
        options.MultipartBodyLengthLimit = 10485760; // 10 MB
    });
}
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.Use(async (context, next) =>
    {
        context.Request.Headers.TryGetValue("Content-Length", out var contentLength);
        if (long.TryParse(contentLength, out var length) && length > 10485760)
        {
            context.Response.StatusCode =
            StatusCodes.Status413PayloadTooLarge;
            await context.Response.WriteAsync("Request size too large.");
            return;
        }
        await next();
    });
}
```

```
app.UseRouting();  
app.UseEndpoints(endpoints =>  
{  
    endpoints.MapControllers();  
});  
}
```

[Result]

If a client sends a request with a body larger than 10 MB, the server responds with a status code 413 Payload Too Large and the message "Request size too large."

The `MultipartBodyLengthLimit` setting is crucial in preventing potential vulnerabilities associated with large file uploads. By setting this limit, you can ensure that the server is not overwhelmed by requests containing excessively large data, which could lead to resource exhaustion or service outages. In this example, a 10 MB limit is enforced, but this value can be adjusted based on the specific needs of your application. The middleware checks the `Content-Length` header and ensures the size does not exceed the configured limit before processing the request further.

[Trivia]

The 413 Payload Too Large status code is part of the HTTP/1.1 protocol. This status indicates that the server is refusing to process a request because the request payload is larger than the server is willing or able to process. It's a best practice to handle large requests in this way to protect the server from abuse.

Using ApiController Attribute in ASP.NET Core

The ApiController attribute in ASP.NET Core is used to automatically enforce conventions and behaviors that are commonly required for building Web APIs, such as model validation and response formatting.

Here is an example of how to apply the ApiController attribute in a Web API controller to automatically handle model validation errors.

[Code]

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase
{
    [HttpPost]
    public IActionResult CreateProduct([FromBody] ProductModel
product)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        // Normally, you'd save the product to a database here
        return Ok("Product created successfully!");
    }
}

public class ProductModel
{
    [Required]
    public string Name { get; set; }
    [Range(0.01, double.MaxValue)]
    public decimal Price { get; set; }
}
```


[Result]

If a client sends a request with invalid data, such as a missing Name or a Price less than or equal to 0, the server automatically returns a 400 Bad Request status with detailed validation errors in the response body.

The ApiController attribute provides several benefits, including automatic model state validation, which means that developers don't need to manually check if the model is valid in every action. This attribute also helps in returning consistent and structured error responses. When a model validation fails, the framework automatically returns a 400 Bad Request with detailed information about the validation errors, which can be immensely helpful for API clients in understanding what went wrong.

[Trivia]

The ApiController attribute also enables automatic binding source inference. This means that the framework can infer where a parameter should come from (e.g., query string, route, body) based on its type and the action method's signature, reducing the need for explicit attributes like [FromBody] or [FromQuery]. However, it's still a good practice to specify these attributes to avoid ambiguity and make the code more readable.

Understanding Cookie Authentication in ASP.NET Core

The `CookieAuthenticationDefaults` class provides default values for cookie-based authentication in ASP.NET Core applications. This class is essential for configuring cookie authentication, which is a common method for maintaining user sessions in web applications.

The following code snippet demonstrates how to set up cookie authentication using the `CookieAuthenticationDefaults` class in an ASP.NET Core application.

[Code]

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthentication(CookieAuthenticationDefaults.Authentica
tionScheme)
        .AddCookie(options =>
        {
            options.LoginPath = "/Account/Login";
            options.LogoutPath = "/Account/Logout";
            options.AccessDeniedPath = "/Account/AccessDenied";
        });
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment
env)
{
    app.UseAuthentication();
    app.UseAuthorization();
}
```

[Result]

No output will be displayed as this code is part of the configuration setup. However, it enables cookie authentication for your application.

In this example, we configure the authentication services in the `ConfigureServices` method. The `AddAuthentication` method specifies the default authentication scheme to be cookie-based. The `AddCookie` method allows us to customize the cookie authentication options, such as defining paths for login, logout, and access denied pages. The `UseAuthentication` middleware must be added in the `Configure` method to ensure that authentication is processed for incoming requests. This setup allows users to be redirected to the login page if they attempt to access protected resources without being authenticated.

[Trivia]

Cookie authentication is widely used in web applications to maintain user sessions. When a user logs in, a cookie is created and sent to the user's browser, which is then sent back to the server with each subsequent request.

The `CookieAuthenticationDefaults.AuthenticationScheme` is a constant that represents the default authentication scheme name, which is "Cookies".

It's important to set up secure cookie options, such as setting `options.SlidingExpiration` to true, which allows the cookie to be renewed on user activity, extending the session duration.

Configuring CORS Policies with CorsPolicyBuilder in ASP.NET Core

ASP.NET Core's CorsPolicyBuilder is used to configure Cross-Origin Resource Sharing (CORS) policies, which control how resources can be requested from another domain. This is crucial for enabling secure interactions between different web applications.

The following code snippet illustrates how to configure CORS policies using the CorsPolicyBuilder in an ASP.NET Core application.

[Code]

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddPolicy("AllowSpecificOrigin",
            builder => builder.WithOrigins("https://example.com")
                               .AllowAnyHeader()
                               .AllowAnyMethod());
    });
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseCors("AllowSpecificOrigin");
}
```

[Result]

No output will be displayed as this code is part of the configuration setup. However, it enables CORS for requests coming from <https://example.com>.

In this example, we define a CORS policy named "AllowSpecificOrigin" in the `ConfigureServices` method. The `WithOrigins` method specifies which domain is allowed to access the resources. The `AllowAnyHeader` and `AllowAnyMethod` methods permit all headers and HTTP methods (GET, POST, etc.) from the specified origin.

The `UseCors` middleware is then added in the `Configure` method, which applies the defined CORS policy to incoming requests. This setup is essential for allowing cross-origin requests while maintaining security.

[Trivia]

CORS is a security feature implemented by web browsers to prevent malicious websites from making requests to another domain without permission.

The `CorsPolicyBuilder` provides a fluent API to build CORS policies, allowing for fine-grained control over which origins, headers, and methods are permitted.

It is advisable to restrict CORS to specific origins rather than using `AllowAnyOrigin`, which can expose your application to security risks.

Understanding PartialView Method in ASP.NET Core

The PartialView method in ASP.NET Core is used to return a partial view as a PartialViewResult. This method is particularly useful when you want to render a portion of a view, often used for reusing common user interface elements like headers, footers, or forms across different views.

Below is an example demonstrating how to use the PartialView method to render a partial view in an ASP.NET Core MVC application.

[Code]

```
// Controller Code
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
    public PartialViewResult GetPartialView()
    {
        var model = new MyModel { Name = "Partial View Example" };
        return PartialView("_MyPartialView", model);
    }
}
// _MyPartialView.cshtml (Partial View)
@model MyModel
<div>
    <h2>@Model.Name</h2>
</div>
// Index.cshtml (Main View)
<div>
    <h1>Main View Content</h1>
    <div id="partialViewContainer">
```

```
        @{Html.RenderAction("GetPartialView");}  
    </div>  
</div>  
// MyModel.cs (Model)  
public class MyModel  
{  
    public string Name { get; set; }  
}
```

[Result]

When the main view Index.cshtml is loaded, the partial view `_MyPartialView` is rendered inside the div with the ID `partialViewContainer`. The output will be: Main View Content
Partial View Example

The `PartialView` method is typically used when you need to render a portion of a view without the need to refresh the entire page. This is particularly useful in scenarios where you need to update a small part of the page based on user interaction, like form submissions, without affecting the rest of the page. The `PartialView` method does not execute the entire MVC pipeline, making it more efficient for rendering smaller segments of a page. In the provided example, `Html.RenderAction` is used to call the `GetPartialView` action method directly from the main view, rendering the partial view `_MyPartialView` with the provided model data. This approach helps in keeping your views DRY (Don't Repeat Yourself) by reusing view components.

[Trivia]

Using `PartialView` differs from `View` in that `PartialView` doesn't execute `_ViewStart.cshtml`, making it a lightweight option for rendering content that doesn't need the full view infrastructure. This can improve performance in situations where full view processing isn't necessary.

Forward Compatibility in ASP.NET Core with Feature Flags

ASP.NET Core supports forward compatibility through the use of feature flags and application settings, allowing developers to enable or disable features dynamically. This approach is valuable when deploying new features incrementally or rolling back features without redeploying the entire application.

The following example illustrates how to implement a feature flag in ASP.NET Core to toggle a feature based on configuration settings.

[Code]

```
// appsettings. {
  "FeatureManagement": {
    "EnableNewFeature": true
  }
}
// Startup.cs
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    services.AddFeatureManagement();
}
// HomeController.cs
public class HomeController : Controller
{
    private readonly IFeatureManager _featureManager;
    public HomeController(IFeatureManager featureManager)
    {
        _featureManager = featureManager;
    }
    public async Task<IActionResult> Index()
    {

```



```

        if (await _featureManager.IsEnabledAsync("EnableNewFeature"))
        {
            ViewData["Message"] = "New Feature is enabled!";
        }
        else
        {
            ViewData["Message"] = "New Feature is disabled.";
        }
        return View();
    }
}
// Index.cshtml
<div>
    <h1>@ViewData["Message"]</h1>
</div>

```

[Result]

Based on the appsettings.json configuration, the output will be:vbnet
New Feature is enabled!
If the EnableNewFeature setting is set to false, the output will be:vbnet
New Feature is disabled.

Feature flags allow for a flexible and controlled rollout of new features in your application. By utilizing the IFeatureManager interface, you can easily check the state of a feature and modify the behavior of your application accordingly. This can be especially useful in continuous integration and continuous deployment (CI/CD) environments, where you might want to deploy new features to specific user groups or test environments without affecting the entire user base. Feature flags also provide a safe way to disable problematic features in production without needing to redeploy the application, reducing downtime and improving the overall stability of the system.

[Trivia]

In ASP.NET Core, feature flags can be combined with environment-specific settings in appsettings.Development.json or appsettings.Production.json to enable or disable features based on the

deployment environment, making it easier to manage feature rollouts across different stages of the application lifecycle.

Using JsonResult to Return JSON Data in ASP.NET Core

JsonResult in ASP.NET Core is a class used to return data in JSON format from a controller action. This is particularly useful in APIs or when you want to send structured data to a client-side application, like a JavaScript frontend.

Below is a simple example where JsonResult is used to return a list of objects in JSON format from an ASP.NET Core controller.

[Code]

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
namespace MyApp.Controllers
{
    public class ProductsController : Controller
    {
        public JsonResult GetProducts()
        {
            var products = new List<object>
            {
                new { Id = 1, Name = "Laptop", Price = 999.99 },
                new { Id = 2, Name = "Smartphone", Price = 499.99 }
            };
            return Json(products);
        }
    }
}
```

[Result]

[

```
[{"Id":1,"Name":"Laptop","Price":999.99},  
 {"Id":2,"Name":"Smartphone","Price":499.99}  
]
```

JsonResult is a convenient way to send JSON-formatted data from your ASP.NET Core application to the client. By default, the JsonResult object will use the JSON serializer to convert the provided object into a JSON string. This is particularly useful in creating RESTful services, where client applications often consume JSON data. When working with APIs, JsonResult can be an easy way to structure and send data. However, in more complex scenarios, you might prefer returning an IActionResult type that can represent different HTTP responses, including JSON. ASP.NET Core automatically serializes the object passed to the Json method using the default JSON serializer, which, as of .NET 5 and later, is System.Text.Json. You can customize this serialization process by modifying settings in JsonOptions. Additionally, the JsonResult class supports asynchronous operations, which is critical when dealing with large datasets or data that requires processing before being sent back as JSON.

[Trivia]

In ASP.NET Core, if you do not explicitly set the content type for a JsonResult, the framework automatically sets it to "application/json". This helps ensure that clients correctly interpret the response as JSON data.

Understanding AssemblyPart in ASP.NET Core MVC

The `AssemblyPart` class in ASP.NET Core is used to discover application parts that make up an MVC application. These parts can include controllers, views, and razor pages, and they are essential for extending the functionality of an MVC application.

Here is an example that demonstrates how to use `AssemblyPart` to load additional assemblies containing MVC controllers.

[Code]

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Mvc.ApplicationParts;
using System.Reflection;
namespace MyApp
{
    public class Startup
    {
        public void ConfigureServices(IServiceCollection services)
        {
            var assembly = Assembly.Load("ExternalAssembly");
            services.AddControllersWithViews()
                .PartManager.ApplicationParts.Add(new
AssemblyPart(assembly));
        }
        public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
        }
    }
}
```

```

    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}
}
}

```

[Result]

If "ExternalAssembly" contains controllers, they will be discovered and used by the MVC application without needing to be directly referenced in the main project.

AssemblyPart is a part of the ASP.NET Core MVC infrastructure that allows you to modularize and extend your application. By using AssemblyPart, you can load controllers, views, and razor pages from external assemblies dynamically. This is particularly useful in large applications where different teams work on different modules, or when you want to create a plugin-based architecture. In the example above, the `PartManager.ApplicationParts.Add(new AssemblyPart(assembly))` line adds an external assembly to the MVC application. This enables the MVC application to discover and use controllers, views, or razor pages defined in that assembly. This approach is highly beneficial when you need to develop scalable applications with a modular architecture. It allows for better separation of concerns and enables teams to work independently on different modules that can be integrated seamlessly. To use AssemblyPart effectively, ensure that the external assembly is correctly referenced and that it contains valid MVC components like controllers or views. Also, be aware that the discovery process of parts

can impact the startup time of your application, especially if dealing with many large assemblies.

[Trivia]

The `AssemblyPart` class is an implementation of the `IApplicationPartTypeProvider` interface. This interface is part of the ASP.NET Core's application part system, which is crucial for discovering and managing the components (such as controllers and views) that make up an MVC application. By customizing this discovery process, developers can significantly extend or modify the behavior of their MVC applications.

Understanding the IWebHostEnvironment Interface in ASP.NET Core

The `IWebHostEnvironment` interface in ASP.NET Core provides essential information about the web hosting environment in which the application is running. It allows developers to access environment-specific properties such as the application's content root path, environment name, and whether the application is running in development or production mode.

The following code snippet demonstrates how to use the `IWebHostEnvironment` interface to retrieve and display the application environment details.

[Code]

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc;
public class HomeController : Controller
{
    private readonly IWebHostEnvironment _env;
    public HomeController(IWebHostEnvironment env)
    {
        _env = env;
    }
    public IActionResult Index()
    {
        return Content($"Environment: {_env.EnvironmentName}, Content
Root: {_env.ContentRootPath}");
    }
}
```


[Result]

Environment: Development, Content Root: /path/to/your/application

In this example, the HomeController receives an instance of IWebHostEnvironment through dependency injection. The Index action method then returns a simple string containing the current environment name and the content root path of the application. The content root path is the base path for the application, which is where the application files are located.

The EnvironmentName property can return values such as "Development", "Staging", or "Production", allowing you to tailor your application's behavior based on the environment. For instance, you may want to enable detailed error messages in development but suppress them in production.

[Trivia]

The IWebHostEnvironment interface was introduced in ASP.NET Core 2.0 and is crucial for managing configurations that vary between different deployment environments.

You can set the environment name in the launch settings or through environment variables, which helps in managing different configurations for various stages of development and deployment.

Exploring the DefaultHttpContext Class in ASP.NET Core

The `DefaultHttpContext` class in ASP.NET Core is the standard implementation of the `HttpContext` class, which encapsulates all HTTP-specific information about an individual HTTP request. It provides access to request and response data, user information, and other contextual data needed during the processing of a request.

The following code snippet illustrates how to create and use an instance of `DefaultHttpContext` to simulate an HTTP request and access its properties.

[Code]

```
using Microsoft.AspNetCore.Http;
public class HttpContextExample
{
    public void ProcessRequest()
    {
        var context = new DefaultHttpContext();
        // Simulating a request
        context.Request.Method = "GET";
        context.Request.Path = "/example";
        context.Response.OnStarting(() =>
        {
            context.Response.StatusCode = 200;
            return Task.CompletedTask;
        });
        // Accessing request and response properties
        var method = context.Request.Method; // "GET"
        var path = context.Request.Path;    // "/example"
        var statusCode = context.Response.StatusCode; // 200
        Console.WriteLine($"Method: {method}, Path: {path}, Status
Code: {statusCode}");
    }
}
```

```
}  
}
```

[Result]

Method: GET, Path: /example, Status Code: 200

In this example, we create an instance of `DefaultHttpContext` to simulate an HTTP request. We set the request method to "GET" and the request path to "/example". The `OnStarting` method is used to set the response status code to 200 when the response is about to be sent.

The properties of `HttpContext` are essential for handling HTTP requests and responses. The `Request` property gives you access to the incoming request details, while the `Response` property allows you to control the outgoing response. This class is fundamental in middleware and controller actions for managing the flow of data in web applications.

[Trivia]

`HttpContext` is a crucial part of the ASP.NET Core pipeline, and understanding how it works is essential for building robust web applications.

The `DefaultHttpContext` class is typically used internally by the ASP.NET Core framework, but it can be instantiated for testing or simulation purposes, as shown in the example.

Afterword



As we reach the conclusion of this journey through essential knowledge for beginners in ASP.NET Core, I want to take a moment to express my gratitude to you, the reader.

Your commitment to enhancing your skills in this powerful framework is commendable, and I hope this book has provided you with the focused insights necessary to deepen your understanding.

This book is designed specifically for those who already possess a foundational grasp of programming concepts.

By concentrating solely on the must-know aspects of ASP.NET Core, it aims to equip you with the tools and knowledge to navigate this framework with confidence.

Whether you are just starting out or are a seasoned developer looking to refresh your knowledge of the latest features, I trust that you have found valuable information within these pages.

Your feedback is incredibly important to me.

If you have enjoyed this book, or if there are areas where you believe it could be improved, I would greatly appreciate your thoughts.

Even if you're short on time, a simple star rating would mean a lot.

I personally read every review, and your insights help shape future editions and topics.

I am always eager to hear what themes or subjects you would like to see explored in future works.

My goal is to provide content that is not only informative but also engaging and relevant to your needs.

Thank you for taking the time to read this book.

I genuinely hope it has been a beneficial resource for your ASP.NET Core journey.

I look forward to connecting with you again in future publications.

Your voice matters, and together, we can continue to explore the exciting world of programming.

Table of Contents

ASP.NET Core is Cross-Platform	6
ASP.NET Core Supports Dependency Injection	8
Understanding Middleware in ASP.NET Core	10
Kestrel: The Default Web Server for ASP.NET Core	12
Razor Pages: A Simpler Alternative to MVC in ASP.NET Core	13
Unified Web API and MVC Framework in ASP.NET Core	15
Understanding Configuration Hierarchy in ASP.NET Core	17
Caching Strategies in ASP.NET Core	19
Entity Framework Core for Database Interactions	21
ASP.NET Core Identity for User Authentication	24
Understanding Tag Helpers in ASP.NET Core	27
Logging in ASP.NET Core with Built-in Support	29
Health Checks in ASP.NET Core for Application Monitoring	31
Environment-Specific Behavior in ASP.NET Core Hosting Environment	33
Hosting Options for ASP.NET Core Applications	36
Understanding Minimal APIs in ASP.NET Core	38
Global Error Handling in ASP.NET Core with UseExceptionHandler Middleware	40
High-Performance Remote Procedure Calls with gRPC in ASP.NET Core	43
Understanding appsettings.json in ASP.NET Core	46
Introduction to Action Filters in ASP.NET Core	48

Data Protection in ASP.NET Core: Encrypting and Decrypting Data	50
SignalR in ASP.NET Core: Real-Time Web Functionality	52
Understanding Model Binding in ASP.NET Core	54
Creating Custom Middleware in ASP.NET Core	56
Customizable Routing in ASP.NET Core	58
Building Web UIs with Blazor and C#	60
Configuring the Request Pipeline with IApplicationBuilder	62
Understanding the IHostingEnvironment Interface	64
Default HTTPS Support in ASP.NET Core	67
Deployment Options in ASP.NET Core	69
Built-in Support for Data Formats in ASP.NET Core APIs	71
Serving Static Files with Middleware	74
Customizing the Dependency Injection (DI) Container in ASP.NET Core	76
Handling Multiple Cultures and Languages with Request Localization in ASP.NET Core	78
Understanding Razor Syntax in ASP.NET Core	81
Preventing CSRF Attacks with Anti-Forgery Tokens in ASP.NET Core	83
Storing Sessions in ASP.NET Core	85
Understanding ControllerBase in ASP.NET Core	87
Middleware Execution Order in ASP.NET Core	89
Understanding IActionResult in ASP.NET Core	91
API Versioning in ASP.NET Core	93
Logging with ILogger in ASP.NET Core	95
Running ASP.NET Core Applications in Docker Containers	97

Understanding ModelState for Input Validation in ASP.NET Core	99
Handling File Uploads with IFormFile in ASP.NET Core	102
Using UserManager for User Operations in ASP.NET Core Identity	104
Introduction to Razor View Engine in ASP.NET Core	106
ASP.NET Core Authentication Middleware Overview	108
Authorization Policies in ASP.NET Core	110
Configuring and Starting a Web Application with WebHostBuilder	112
Custom Route Constraints in ASP.NET Core	114
Using HttpClientFactory in ASP.NET Core	116
Using GraphQL with ASP.NET Core for Flexible APIs	118
Building Configuration Sources with ConfigurationBuilder in ASP.NET Core	121
Using Environment Variables in ASP.NET Core	123
JWT Bearer Tokens for API Authentication in ASP.NET Core	125
Understanding Dependency Injection Scopes in ASP.NET Core	127
Mastering Model Validation in ASP.NET Core	130
Accessing Application Settings with IConfiguration in ASP.NET Core	133
Handling Exceptions with Custom Exception Filters in ASP.NET Core	135
Understanding View Components in ASP.NET Core	138
Using ObjectResult in ASP.NET Core	140
Understanding the IFormCollection Interface in ASP.NET Core	142
CORS Support in ASP.NET Core	144

Using TempData in ASP.NET Core	147
Generating URLs with IUrlHelper in ASP.NET Core	149
Configuring Session State in ASP.NET Core	151
Adding Real-Time Functionality with ASP.NET SignalR	154
Response Compression in ASP.NET Core	157
Database Configuration by Environment in ASP.NET Core	160
Understanding Endpoint Routing in ASP.NET Core	163
Managing Forms with ASP.NET Core's FormTagHelper	166
File Streaming in ASP.NET Core Using FileStreamResult	169
Configuring Cookies with ASP.NET Core's CookieOptions	171
Configuring Custom Razor View Locations in ASP.NET Core	173
Understanding the ActionResult Class in ASP.NET Core	175
Dynamic Compilation of Razor Views in ASP.NET Core	177
Implementing Custom Logic with IActionFilter in ASP.NET Core	179
Understanding IFormFileCollection in ASP.NET Core	181
Standardizing API Responses with API Conventions in ASP.NET Core	183
Custom Authorization with AuthorizationHandler in ASP.NET Core	185
Global Exception Handling with Middleware in ASP.NET Core	188
Razor Pages Simplify Page-Focused Development in ASP.NET Core	191

Using StatusCodes Class for HTTP Status Codes in ASP.NET Core	193
Using IApplicationLifetime for Application Startup and Shutdown in ASP.NET Core	195
Enabling Request Body Buffering in ASP.NET Core	198
Understanding ActionResult<T> in ASP.NET Core	201
Using HttpContext in ASP.NET Core	203
Configuring Form Data Processing with ASP.NET Core's FormOptions Class	205
Passing Data to Views Using ViewData in ASP.NET Core	207
Graceful Shutdown in ASP.NET Core	209
Generating URLs with Url.Action in ASP.NET Core	211
Limiting Request Size in ASP.NET Core	213
Using ApiController Attribute in ASP.NET Core	215
Understanding Cookie Authentication in ASP.NET Core	217
Configuring CORS Policies with CorsPolicyBuilder in ASP.NET Core	219
Understanding PartialView Method in ASP.NET Core	221
Forward Compatibility in ASP.NET Core with Feature Flags	223
Using JsonResult to Return JSON Data in ASP.NET Core	226
Understanding AssemblyPart in ASP.NET Core MVC	228
Understanding the IWebHostEnvironment Interface in ASP.NET Core	231
Exploring the DefaultHttpContext Class in ASP.NET Core	233